Parallel Programming: Orchestration

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

© C. Xu 1999
Grid Solver Example

Expression for updating each interior point:

- Simplified version of solver in Ocean simulation

```
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 (an (n+2)x(n+2) array);
7. initialize(A); /*initialize 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 inter. points*/
18.     for j ← 1 to n do
19.       temp = A[i,j]; /*save old value of element*/
22.       diff += abs(A[i,j] - temp);
23.     end for
24.   end for
25.   if (diff/(n*n) < TOL) then done = 1;
26. end while
27. end procedure

Decomposition

• Simple way to identify concurrency is to look at loop iterations
  —dependence analysis; if not enough concur., then look further
• Not much concurrency here at this level (all loops sequential)
• Examine fundamental dependences, ignoring loop structure
Assignment

• Static assignments (given decomposition into rows)
  - block assignment: Row $i$ is assigned to process $\frac{i}{p}$

  
  
  
  Cyclic assignment: Process $i$ is assigned rows $i$, $i+p$, $i+2p$

Block cyclic:

Orchestration: Data Parallel Solver

1. int n, nprocs;
   /* grid size $(n+2)\times(n+2)$ 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); /* initialize the matrix A somehow*/
8.   Solve (A); /* call the routine to solve equation*/
9.   end
10. procedure Solve(A) /* solve the equation system*/
11.   …
27. end procedure
Data Parallel Solver

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 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];
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

Orchestration: Shared Address Space Solver

Single Program Multiple Data (SPMD)

• Assignment controlled by values of variables used as loop bounds
1. int n, nprocs;
2a. float **A, diff; /*A is global array representing the grid*/
   /*diff is global (shared) maximum difference in current sweep*/
2b. LOCKDEC(diff_lock);
2c. BARDEC(bar1); /*barrier synchronization between sweeps*/
3. main()
4. begin
5. read(n); read(nprocs);
6. A ← G_MALLOC (a n+2 x n+2 double array);
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

10. procedure Solve(A)
11. float **A; /*A is entire n+2-by-n+2 shared array*/
12. begin
13. int i, j, pid, done = 0;
14. float temp, mydiff = 0; /*private variables*/
14a. int mymin = 1 + (pid * n/nprocs);
14b. int mymax = mymin + n/nprocs – 1
15. while (!done) do
16. mydiff = diff = 0;
16a. BARRIER(bar1, nprocs); 
17. for i ← mymin to mymax do /*for all interior rows*/
18. for j ← 1 to n do /*for all interior elements*/
19. temp = A[i,j];
21. mydiff += abs(A[i,j] - temp);
22. endfor
23. endfor
24. LOCK(diff_lock); /*update global diff if necessary*/
25. diff += mydiff;
25a. UNLOCK(diff_lock);
25b. BARRIER(bar1, nprocs);
25c. if (diff/(n*n) < TOL) then done = 1;
25d. BARRIER(bar1, nprocs);
26. endwhile
27. end procedure
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?

© C. Xu 1999

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

- 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 excl.

- 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

<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><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
</tr>
<tr>
<td>solve eqn system</td>
<td>solve eqn system</td>
<td>solve eqn system</td>
</tr>
<tr>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
</tr>
<tr>
<td>apply results</td>
<td>apply results</td>
<td>apply results</td>
</tr>
<tr>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
<td><strong>Barrier</strong> (name, <em>nprocs</em>)</td>
</tr>
</tbody>
</table>

- 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>b:</td>
</tr>
<tr>
<td></td>
<td>flag</td>
</tr>
<tr>
<td>a:</td>
<td>while (flag is 0) do nothing;</td>
</tr>
<tr>
<td></td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>print A;</td>
</tr>
</tbody>
</table>

*Busy-waiting or spinning*

Orchestration: 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 different
  - data structures and data access/naming
  - communication
  - synchronization
1. int pid, n, b; /* process id, dimension and number of processors to be used*/
2. float **myA;
3. main()
4. begin
5. read(n); read(nprocs);
8a. CREATE (nprocs-1, Solve);
8b. Solve(); /*main process becomes a worker too*/
8c. WAIT_FOR_END (nprocs-1);
 /*wait for all child processes created to terminate*/
9. end main

10. procedure Solve()
11. begin
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*/
19. temp = myA[i,j];
22. mydiff += abs(myA[i,j] - temp);
23. endfor
24. endfor
 /*communicate local diff values and determine if done; can be replaced by reduction and broadcast*/
25. — —
26. endwhile
27. end procedure
15. while (!done) do
16. mydiff = 0;
    /* 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. endif
25m. endif
26. endwhile
27. end procedure

© C. Xu 1999

---

**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 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

© C. Xu 1999
Notes on Message Passing Program

/*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

Send/Receive
  Synchronous  Asynchronous
    Blocking async.  Nonblocking async.

• 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 async. messages
  – With synch. messages, our code is deadlocked. Fix?
**Synchronous versus Asynchronous**

- Synchronous SEND: returns control to the calling process only when it is clear that the corresponding RECEIVE has been performed
- Synchronous RECEIVE: returns control when the data has been received into the destination process’s address space
- Asynchronous: Blocking vs Nonblocking
  - Blocking asynchronous SEND: return control to the calling process when the msg has been copied into system address space
  - Blocking asynchronous RECEIVE: Similar to synchronous RECEIVE, but a blocking RECEIVE does not send an acknowledgement to the sender
  - Non-blocking SEND: return control immediately
  - Non-blocking RECEIVE: return control after simply posting the intent to RECEIVE

---

**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 ...