Chapter 4: Parallel Program Structures IV

Elements of Parallel Computing

Eric Aubanel

Master-Worker

- Good for load balancing
- Master queues tasks (may be implicit, i.e., tasks known)
- Workers de-queue tasks
- Master can also act as worker
- *distributed memory*: master sends tasks and gathers results
- *shared memory*: need to avoid race conditions, for conflicts between master and workers or between workers

Shared Memory Master-Worker Fractal

shared *kount*, *chunkCount*
if $id = 0$ **then**
    $chunkCount \leftarrow nt * chunk$
end
$istart \leftarrow id * chunk$
$iend \leftarrow (id + 1) * chunk - 1$
`barrier()`
while $istart \leq n - 1$ **do**
    for $i \leftarrow istart \ to \ iend$ **do**
        `...// see sequential Algorithm` **??**
    end
    begin critical
        $istart \leftarrow chunkCount$
        $chunkCount \leftarrow chunkCount + chunk$
    end critical
    $iend \leftarrow \min(istart + chunk - 1, n - 1)$
end

Master-Worker Considerations

- Need to choose good task (chunk) size for workers
- Master can get overloaded
- Alternatives include multiple masters, work stealing

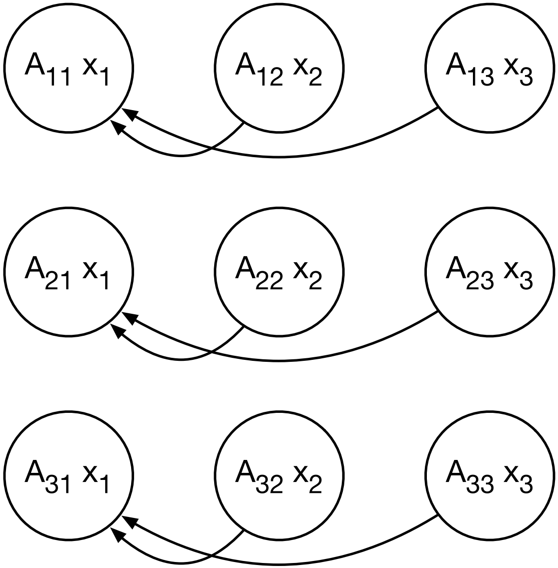
Distributed Memory Programming

- Distributed arrays
- Message passing
- Local and global communication

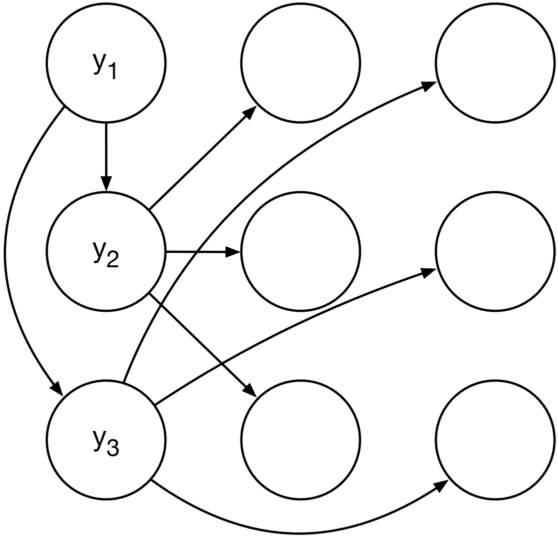
Distributed Arrays: Matrix-Vector Multiplication

$$
\left[\begin{array}{c:c:c} A_{11} & A_{12} & A_{13} \\ \hdashline A_{21} & A_{22} & A_{23} \\ \hdashline A_{31} & A_{32} & A_{33} \end{array}\right] \times \left[\begin{array}{c} x_1 \\ \hdashline x_2 \\ \hdashline x_3 \end{array}\right] = \left[\begin{array}{c} A_{11}x_1 \\ \hdashline A_{21}x_1 \\ \hdashline A_{31}x_1 \end{array}\right] + \left[\begin{array}{c} A_{12}x_2 \\ \hdashline A_{22}x_2 \\ \hdashline A_{32}x_2 \end{array}\right] + \left[\begin{array}{c} A_{13}x_3 \\ \hdashline A_{23}x_3 \\ \hdashline A_{33}x_3 \end{array}\right]
$$

Task Graph



Redistribution of Results



Message Passing

SPMD, with local and global communication

Local Communication

Two-sided: both sender and receiver participate.
E.g., exchanging values:

if $id = 0$ **then**
    nonblocking send data to 1
    receive data from 1
else
    nonblocking send data to 0
    receive data from 0
end

Will this work?

Local Communication

Send/receive: blocking/non-blocking, synchronous/asynchronous

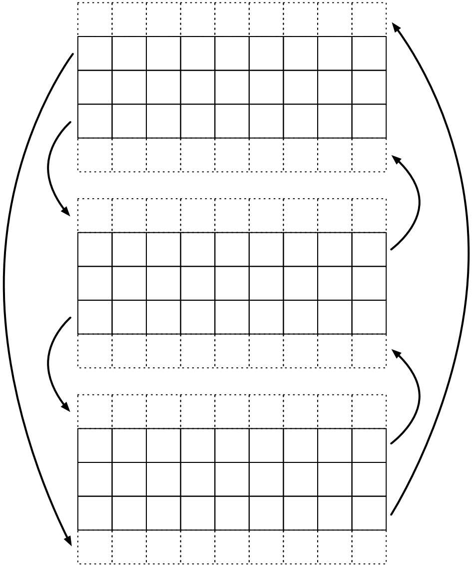
- blocking:
  - synchronous: waits until message received
  - asynchronous: waits until send buffer can be overwritten
- non-blocking: don't wait. Need method to determine when

Potential Deadlock

if $id = 0$ **then**
  blocking send data to 1
  receive data from 1
else
  blocking send data to 0
  receive data from 0
end

Deadlock if communication synchronous.

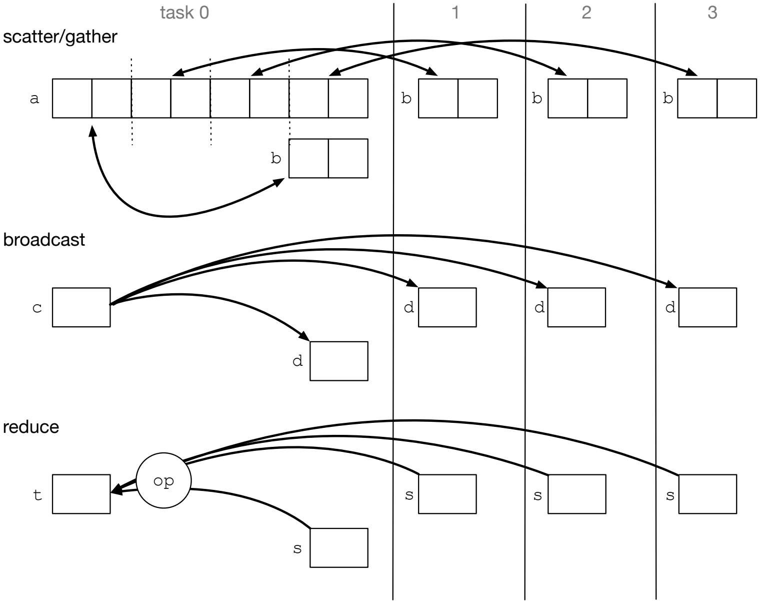
Game of Life



```
// each task has (n/p + 2) × n arrays grid and
    newGrid
Input: n × n grid of cells, each with a state of alive (1) or dead
       (0).
Output: evolution of grid for a given number of generations

nbDown ← (id + 1) mod p, nbUp ← (id − 1 + p) mod p
m ← n/p // Assume n mod p = 0
for a number of generations do
    // nonblocking send of boundary values to
       neighbors
    nonblocking send grid[m, 0..n − 1] to nbDown
    nonblocking send grid[1, 0..n − 1] to nbUp
    // receive boundary values from neighbors into
       ghost elements
    receive from nbDown into grid[m + 1, 0..n − 1]
    receive from nbUp into grid[0, 0..n − 1]
    foreach cell at coordinate (i, j) ∈ (1..m, 0..n) do
        updateGridCell(grid, newGrid, i, j)
    end
```

Global Communication



Scatter/Gather

- `scatter(0,a,2,b)`: source task, source data, number of elements, destination data
- `gather(0,b,2,a)`: destination task, source data, number of elements, destination data

Improving Game of Life

Reading and scattering grid for Game of Life:

if $id = 0$ **then**
  read *grid* from disk
end
`scatter`$(0, grid, n * n/p, grid[1..n/p, 0..n-1])$

Gathering and displaying grid:

if *number of generations* mod $d = 0$ **then**
  `gather`$(0, grid[1..n/p, 0..n-1], n * n/p, dgrid)$
  if $id = 0$ **then**
    display *dgrid*
  end
end

Broadcast/Reduction

- `broadcast(0,c,size,d)`: source task, source data, size of data, destination data
- `reduce(0,s,size,op,t)`: destination task, source data, size of data, destination data

Can perform global communication over subset of tasks

- `broadcast(0,c,size,d, group)`
- `reduce(0,s,size,op,t, group)`

Row-Wise Matrix-Vector Multiplication

$nb \leftarrow n/p$ // `assume` $n \bmod p = 0$
for $i \leftarrow 0$ *to* $nb - 1$ **do**
    $c[i] \leftarrow 0$
    foreach *column* j *of* a **do**
        $c[i] \leftarrow c[i] + a[i,j] * b[j]$
    end
end
`gather`$(0, c, nb, c)$
`broadcast`$(0, c, n, c)$

2D Matrix-Vector Multiplication

```
// Assume p = q^2, matrix is square, n mod q = 0
```

$q \leftarrow \sqrt{p}$, $nb \leftarrow n/q$

for $i \leftarrow 0$ *to* $n/q - 1$ **do**

  $c[i] \leftarrow 0$

  for $j \leftarrow 0$ *to* $n/q - 1$ **do**

    $c[i] \leftarrow c[i] + a[i, j] * b[j]$

  end

end

$rowID \leftarrow \lfloor id/q \rfloor$, $colID \leftarrow id \bmod q$, $destID \leftarrow rowID * q$

$group \leftarrow [destID..destID + q - 1]$

`reduce`($destID$, c, nb, sum, c, $group$)

$sourceID \leftarrow colID * q$

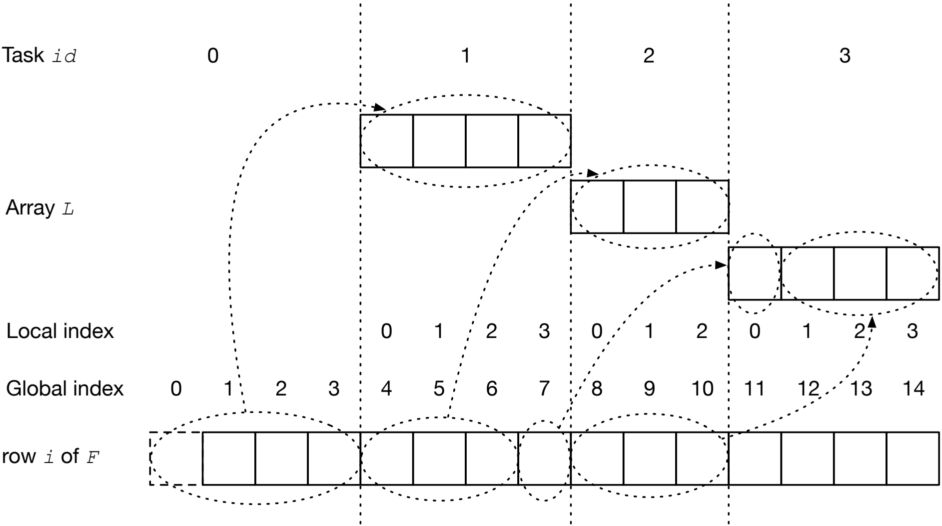
$group \leftarrow [sourceID, colID, colID + q, .., coldID + (q - 1) * q]$

`broadcast`($sourceID$, c, nb, c, $group$)

$group \leftarrow [0, q, .., (q - 1) * q]$

`broadcast`(0, c, nb, c, $group$)

Subset Sum



Task id
0
1
2
3
Array L
Local index
0 1 2 3 0 1 2 0 1 2 3
Global index
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
row i of F

Input: Array $s[1..n]$ of n positive integers, target sum S
Output: Completed array F
Data: Array $F[1..n, 0..nb-1]$ initialized to 0 (nb is number of columns owned by task), array $L[0..\lceil S/p \rceil - 1]$ to store received messages

`broadcast`(0,s,n,s)
$myFirst \leftarrow \lfloor id * S/p \rfloor + 1$

$myLast \leftarrow \lfloor (id+1) * S/p \rfloor$

if $id = 0$ **then** `//first block has one extra value`

  $myFirst \leftarrow 0$
  $F[1,0] \leftarrow 1$
end
$nb \leftarrow myLast - myFirst + 1$
$nL \leftarrow \lceil S/p \rceil$
if $id =$ `findID`($s[1]-1$, p, S) **then**
$F[1, s[1] - myFirst] \leftarrow 1$

```
for i ← 2 to n do
    id1 ← findID(myFirst + s[i] − 1, p, S)
    id2 ← findID(myLast + s[i] − 1, p, S)
    if id1 < p then
        // send ...
    end
    if id > 0 ∧ myLast − s[i] ≥ 0 then
        // receive ...
    end
    solveRow(F, L, s, nb, myFirst, i)
end
// Trace F to return subset, or have task p − 1
   print F[n, myLast] to return yes/no
```

Send

```
if id1 < p then
    if id1 = id2 then
        myLocalBegin ←
        max(0, ⌊id1 * S/p⌋ + 1 − s[i] − myFirst)
        send F[i − 1, myLocalBegin..nb − 1] to id1
    else
        destBegin ← myFirst + s[i]
        destLast ← ⌊(id1 + 1) * S/p⌋
        nb1 ← destLast − destBegin + 1 // # of elements
            to send to id1
        if id1 > id then send F[i − 1, 0..nb1 − 1] to id1
        if id2 < p then  send F[i − 1, nb1..nb − 1] to id2
    end
end
```

Receive

if $id > 0 \wedge myLast - s[i] \geq 0$ **then**

$id1 \leftarrow findID(myFirst - s[i] - 1, p, S)$
if $myFirst - s[i] < 0$ **then**

$myLocalBegin \leftarrow s[i] - myFirst$

else

$myLocalBegin \leftarrow 0$

end
receive from $id1$ into $L[myLocalBegin..nL - 1]$

$nS \leftarrow$ size of message received
$id2 \leftarrow \texttt{findID}(myLast - s[i] - 1,\ p,\ S)$
if $id1 \neq id2 \wedge id2 < id$ **then** **receive** from $id2$ into $L[nS..nL - 1]$

end

```
// return rank of task that owns column j of array
   of length n in a decomposition into p blocks
Procedure findID(j, p, n)
    return ⌊(p * (j + 1) − 1)/n⌋
end
```

```
// Solve row i of F with array L for values
   needed from other tasks
Procedure solveRow(F, L, s, nb, myFirst, i)
    if id = 0 then
        F[i,0] ← 1
        jstart ← 1
    else
        jstart ← 0
    end
    for j ← jstart to nb − 1 do
        F[i,j] ← F[i − 1,j]
        offset ← j − s[i]
        if offset ≥ 0 then //dependency in my array F
            F[i,j] ← F[i,j] ∨ F[i − 1,offset]
        else if offset + myFirst ≥ 0 then //dependency in
        array L
            F[i,j] ← F[i,j] ∨ L[j]
        end
    end
```