MPI Programming

Sanjay Rajopadhye
Colorado State University
Fall 2011 Week 11

Objectives

- MPI Collective Communications
- "Wavefront parallelization"
  - MPI implementation
  - OpenMP
  - SPMD OpenMP

Collective Communications

- Better abstraction for parallel program design and implementation
- Better performance when applications can exploit it
- Somewhat more effort for implementers
- Some applications may require more "low level" communications (e.g., P2P send-recv)

Basic Collectives

- mpi-forum.org/docs/mpi22-report/node86.htm
- Barrier
- Broadcast, Reduce, AllReduce
- Scatter, Gather, AllGather
- All-to-All Scatter/Gather
- ReduceScatter
- Scan (prefix)

Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

- blocks the caller until all group members have called it.
- returns at any process only after all processes in the group enter the call.

Broadcast Reduce & AllReduce

```
int MPI_Bcast(buffer, count, type, root, comm)
int MPI_Reduce(sendbuf, recvbuf, count, type,
   op, root, comm)
```

- Reduce comes in many flavors:
  - Simple reduction
  - All reduce
  - Scan

Reduce Operators

- Predefined or user defined
  - max, min, sum, prod, land, band, lor, bor, lxor, bxor,
  - maxloc, minloc
  - Last two are special. Computes the max/min as well as the index (location) of the result (in case of conflict, smaller index wins)

Scatter & Related Functions

```
int MPI_Scatter(
   void* sendbuf,              /* buffer in each process */
   int sendcount,              /* number of items */
   MPI_Datatype sendtype,
   void* recvbuf,              /* buffer in root process */
   int recvcount,              /* number of items */
   MPI_Datatype recvtype,      /* they can be different */
   int root,
   MPI_Comm comm)
```

Gather, AllGather, & AlltoAll

- Same signature as basic scatter
- Gather does the communication in the opposite order (src and dst are interchanged)
- AllGather also does a broadcast
- All to All is multiple simultaneous scatter-gathers

Illustration

Wavefront Parallelization

- How to parallelize computations (loops) that have apparent dependences
  - None of the loops have completely independent operations

Simple examples

```
for (i=1; i<N; i++)
    for (j=1; j<M; j++)
        A[i,j] = foo(A[i,j-1], A[i-1,j])

for (i=1; i<N; i++)
    for (j=1; j<M; j++)
        B[j] = foo(B[j-1], B[j])

for (i=1; i<N; i++)
    for (j=1; j<M; j++)
        C[i] = foo(C[i-1], C[i])
```

Iteration space & data space

- Iteration space: set of values that the loop iterators can take:
  - Rectangular region bounded by [1,1] and [N-1,M-1]
- Data space: set of legal values of array indices:
  - 2-dimensional "table" in Ex 1
  - 1-D array bounded by [0,M-1] in Ex 2
  - 1-D array bounded by [0,N-1] in Ex 3

References and Dependences

- **Reference**: an occurrence of an array variable on either
  - left hand side (write reference)
  - or right hand side (read reference)

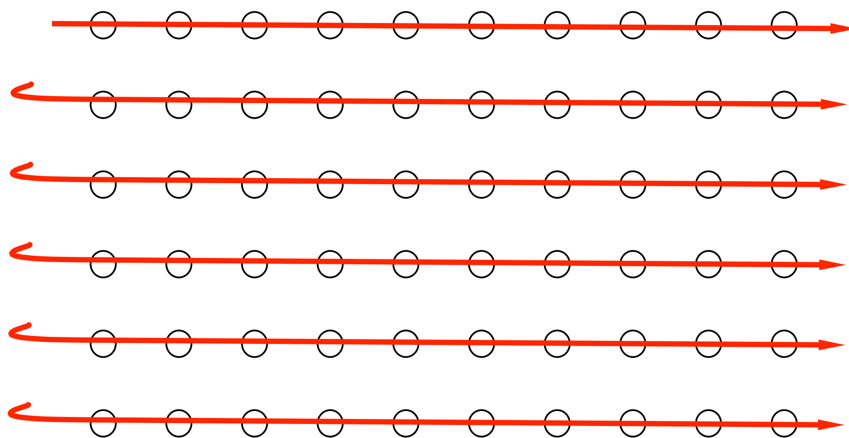
of a statement in the loop body

- **Dependences**: specify which iteration point depends on which others

Finding the Dependences

- Very hard problem (undecidable) in general, but we have a special case
  - Iteration point reads a memory location
  - (Many) iteration points (may) write to the same location
  - Find the "most recent write, for a given read."

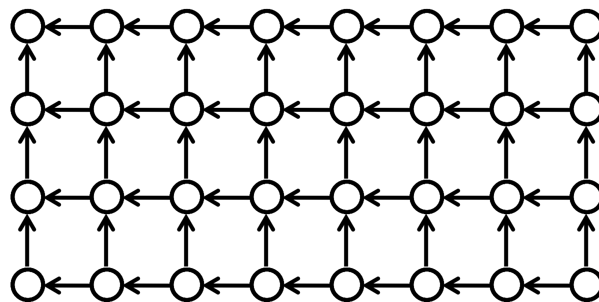
Sequential Execution order



Example solutions

- Example 1 & 2 (same solution): [i,j] depends on
  - [i,j-1] and [i-1,j] (west and north neighbors)
  - Example 2 has an additional (memory-based) dependence: [i+1,j-1] reads the same memory location that [i,j] (over) writes.
- Example 3: [i,j] depends on
  - [i,j-1] and [i-1,M-1]
  - No parallelization possible (midterm example)

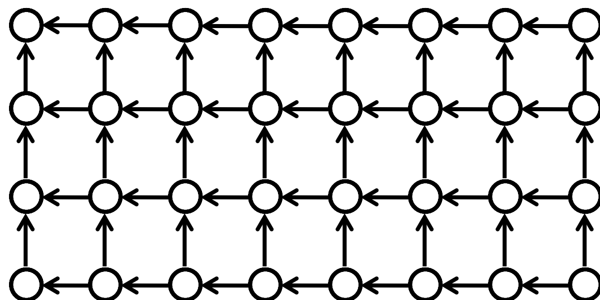
Wavefront Parallelization



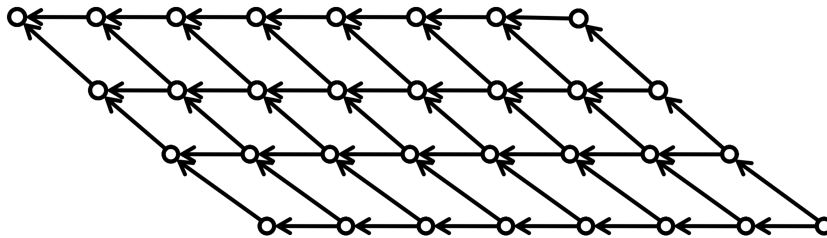
Wavefront Parallelization

- In a task graph, a node can be computed as soon as all its predecessors have been
- Assign time-stamps to each node
- Build a pipeline:
  - Shift each row by 1, relative to its predecessor
  - also called "skewing"
  - Makes a parallelogram out of a rectangle

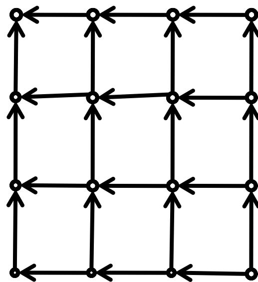
OpenMP Parallelization

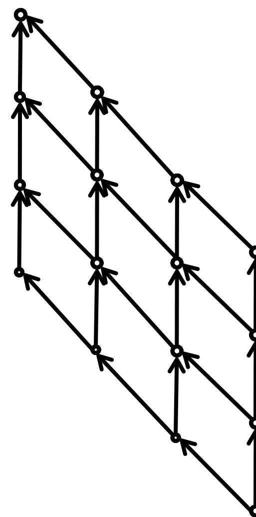


OpenMP Parallelization



Alternative (?) Parallelization





OpenMP Parallelization

- Wavefront parallelization creates a parallelogram from the rectangle:
  - [i,j] is mapped to [t,p] = [i+j-1,j]
  - A boundary i = const becomes t = p+const'
  - A boundary j= const remains unchanged
- Write a loop to visit all the points [t,p] in the parallelogram

OpenMP Parallelization

- At each point determine the
  - computation of the *original iteration point* that was mapped to [t,p]

    [t,p]=[i+j-1,j], solve for i and j: i=t-p+1, j=p
  - memory location the *original iteration point* wrote to
  - add synchronization
  - use additional memory if needed

OpenMP Code skeleton

```
for (t=1; t<(N+M-2); t++){
   #pragma omp parallel for
    for (p=max(1,t-N-2); p<min(t,M-1); p++)
        NEW LOOP BODY
    sync/memory corrections
}
```

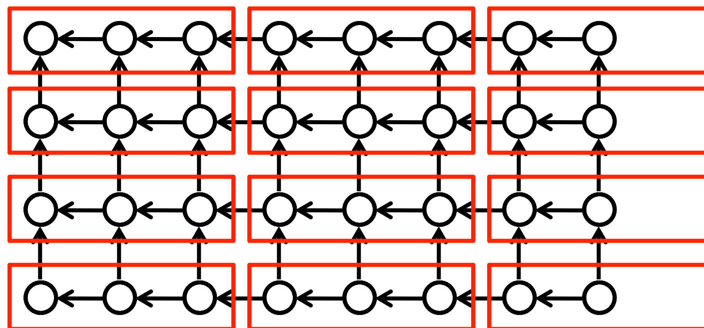
OpenMP Code skeleton

- Control is the same for both examples
- Body for Examples 1 and 2:
  - `A[t-p+1,p] = foo(A[t-p+1,p-1],A[t-p+1,p]);`
  - `Curr[p] = foo(Prev[p],Prev[p-1]);`

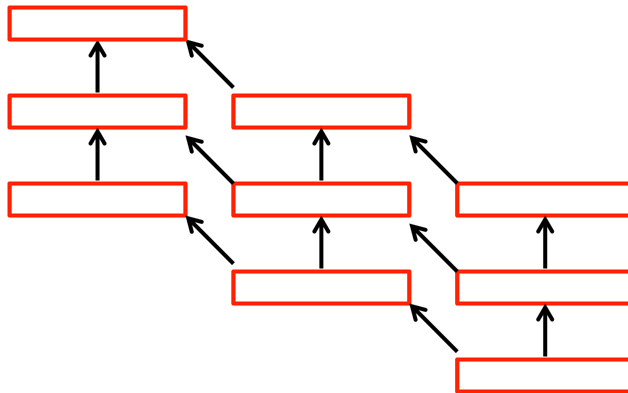
MPI Parallelization (Ex 1&2)

```
for (i=1; i<N; i++)
    for (j=1; j<M; j++)
        A[i,j] = foo(A[i,j-1], A[i-1,j])
```

MPI Parallelization



MPI Parallelization



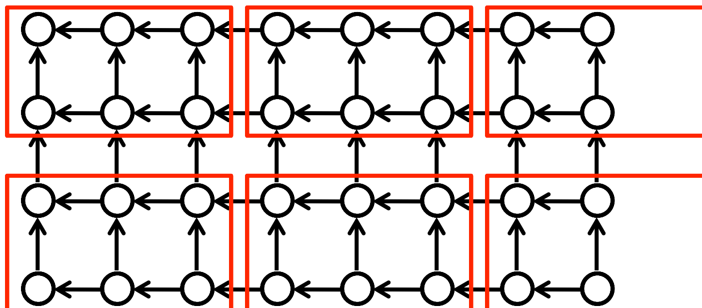
MPI Code skeleton

```
for (i=1; i<N; i++){
    recv from left into x;
    A[i,0] = foo(x,A[i-1,0]);
    for (j=1; j<size-1; j++)
        A[i,j] = foo(A[i,j-1], A[i-1,j]);
    send A[i,size-1] to right;
}
```

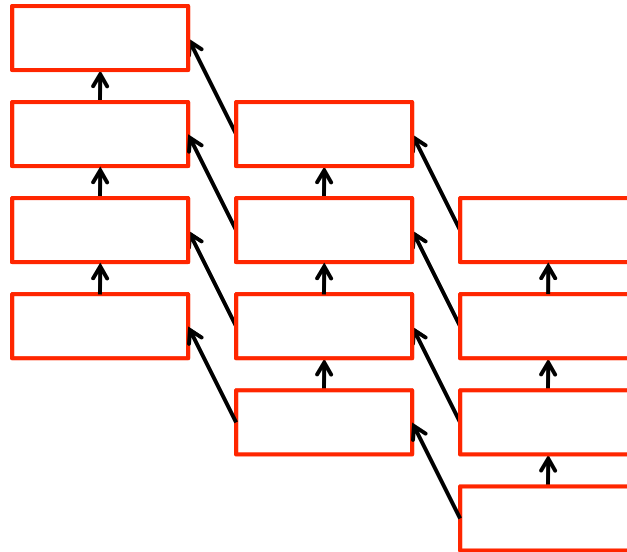
Limitations

- No control of "granularity"
  - In MPI, each process has N communication calls
  - In OpenMP, $N+M$ synchronizations
- Improve granularity
  - In MPI: merge multiple messages into one
  - In OpenMP: use tiling
- Generic Solution: SPMD style code

Coarse-grain MPI



aka Message Vectorization



Coarse grain MPI skeleton

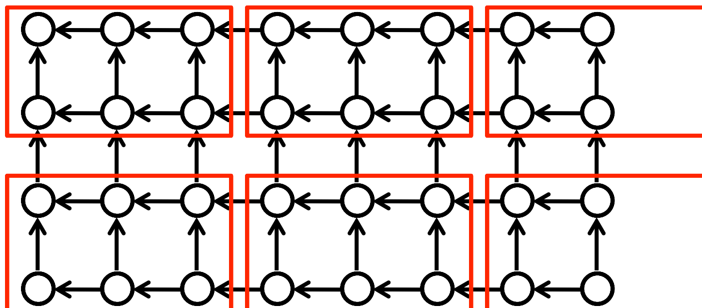
```
for (ii=1; ii<N; ii+=BkSz){                //iterate over the blocks
    recv from left into X;
    for (i=ii*BkSz; i<(min(N,(ii+1)*BkSz-1);i++ // rows in block
    B[0] = foo(X[##],B[0]); //first column in block
    for (j=1; j<size; j++)
        B[j] = foo(B[j-1], B[j]);
     Y[##] = B[size-1] // copy last column to send buffer
    send Y to right;
}
```

is local address in buffer for i-th row in the block, i.e., (i-ii*BkSz)

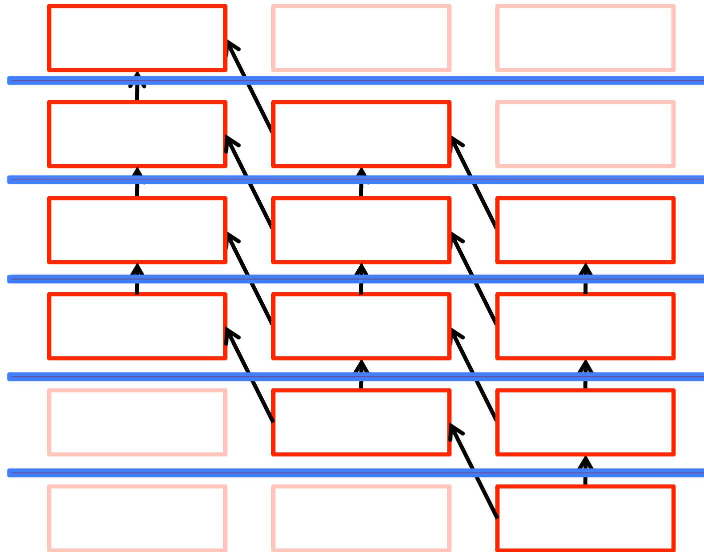
Coarse grain OpenMP

- All communication is implicit (read/write of data arrays in shared memory)
- Processes have some private data (e.g., B) and some shared (e.g., X, Y)
- Barrier synchronization = send-recv
- Dummy iterations

Coarse-grain OpenMP



Dummies & Synchronization



OpenMP skeleton

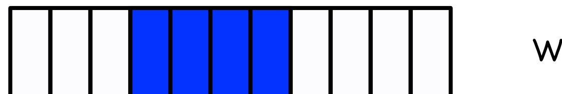
```
p = omp_get_num_threads();
#pragma omp parallel private(tid, start ...)
tid = omp_get_thread_num();
start = tid*(M/p); // global address in B array
for (i=0; i<tid; i++)
#pragma omp barrier // the dummy tiles prolog
/* Main Computation */
for (i=tid; i<p; i++)
#pragma omp barrier // the dummy tiles epilog
```

Main Computation

R and W are global 2D arrays: outer index is (related to) time step and inner one is block size BkSz

```
for (ii=1; ii<N; ii+=BkSz){   //iterate over the blocks
  for (i=ii*BkSz; i<(min(N,(ii+1)*BkSz-1); i++){ //rows in block
    B[start] = foo(R[xx][##],B[start]); // first column reads R
    for (j=1; j<Hsize; j++) //all other columns read previous B
      B[start+j] = foo(B[start+j-1], B[start+j]);
    W[xx][##] = B[start+size-1] //last column is copied to W
    }
# pragma omp barrier
update xx and R, W
}
```

Managing the R, W buffers

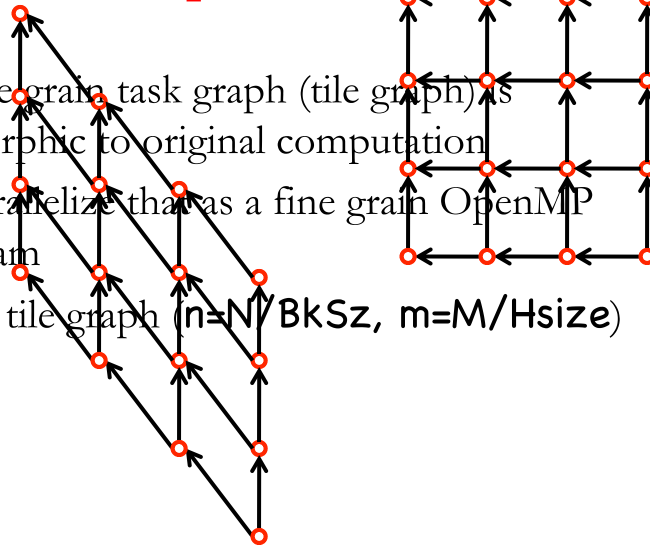


Buffer Management

```
/* Main Computation */
/* Initialize x */
  x=0;  //Is this correct?  Why? Fix if needed
/* Execute the tile body, reading from R[x][i],
   computing the row of B within the tile and
   writing to W[x][i]  */
#pragma omp barrier // do the "send-recv"
#omp pragma single
{tmp=R, R=W; W=tmp;} // swap R and W
x = (x==0)?p-1:x-1 // update x
#pragma omp barrier // "commit" the swap
```

Alternate OpenMP

- Coarse grain task graph (tile graph) is isomorphic to original computation
- So parallelize that as a fine grain OpenMP program
- n x m tile graph (n=N/BkSz, m=M/Hsize)



OpenMP skeleton

```
for (t=1; t<n+m-2; t++){
  // bookkeeping
  #pragma omp parallel private(...)
  for (p=max(1,n-t-1); p<min(t,m); jj++) {
       // Compute block # (t-p-1,p)
       // in the original code as a
       // doubly nested loop
       bookkeeping for buffer management
  }
```

Another Alternate

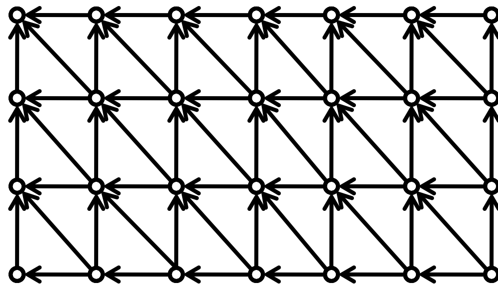
- Start with fine grain parallel code
- Modify the
  - loop bounds, and
  - stride

  so that only multiples of BkSz and Hsize (i.e., tile origins) are visited by the loop
- At iteration point [ii,jj] execute that block of the original iteration space

More Examples

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    Tab[i,j] = foo(Tab[i-1,j], Tab[i,j-1], Tab[i-1,j-1]);
```

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    X[j] = foo(X[j-1], X[j], X[j+1]);
```



- Dependence analysis to find the dependences
- Agglomeration = tiling
- Wavefront parallelism can be applied at many levels (fine grain, coarse grain) and in many forms
  - Fine/coarse grain
  - MPI and/or OpenMP