Distributed Memory Parallel Programming and MPI

Kai Shen

Parallel Programming Model

- Shared memory
  - Writes to a shared location are visible to all
- Distributed memory
  - No shared memory; data is partitioned and must be shared through explicit communication (message passing)
- Problems
  - Additional challenges to programmers: explicit communications, data partitioning
  - Slower inter-process communications
- Why distributed memory parallel programming?

Why Distributed Memory Parallel Programming?

- Less requirement on hardware support
  - cluster of machines connected by commodity network
  - better take advantage of newest processors
- More portable
  - can run on shared memory multiprocessors
  - can run on hybrid distributed/shared-memory platforms
- No worry of multiprocessor memory consistency
  - no memory consistency-related errors
- Better locality of parallel programs
  - programmer knows her/his program best

Parallel Programming Steps

Converting a sequential application to a parallel one

- Decomposition into tasks
- Assign tasks to processors
  - Also partition the data
- Orchestrate data access and synchronization
  - Orchestrate message passing and synchronization

Message Passing Interface

- De facto standard programming interface for message passing-based parallel programs
  - think of pthread for shared memory parallel programming
- You write a single program, multiple copies of which will run on multiple processors
  - Assumption: all processes do mostly similar things
  - Different parts distinguish through process ID
- Communications
  - Point-to-point: send/receive
  - Group communications: broadcast, gather, scatter, reduce, barrier
- It is a programming interface, not an implementation specification!

MPI Send/Receive

- Matching send/receive:
  - Process x sends a message to process y
  - Process y receives a message from process x
- Nonblocking receive
- Nonblocking send
- Synchronous send
- … … … …

MPI Group Communications

- Barrier
  - All processes wait until all have arrived
- Broadcast
  - One process (root) sends a message to be received by others
- Reduce
  - A function (MAX, SUM, …) is applied to data supplied by all processes; result is returned at one process (root)
  - Function is evaluated following process rank order
    - Can be optimized if associative and/or commutative
- … … … …
- Can operate in a sub-set of processes
  - Customized MPI communicator

Performance Issues

- Load balance (dynamic task assignment difficult)
  - ⇨ careful data partition and task assignment
- Synchronization/communication wait
  - ⇨ block as late as possible
- Long communication latency (vs. high bandwidth)
  - our Ethernet cluster: 250us latency, 80MB/sec bandwidth if 1KB per synchronization, effective bw is 4MB/sec
  - ⇨ synchronize/wait as few times as possible
- Group communication lead to group wait
  - ⇨ only when necessary, on smallest group necessary

Deadlocks

- Why does my MPI program gets stuck?
- Possible reasons
  - Receive without a matching send (or a matching send cannot be reached)
  - Group communications are not called by all in the group
  - Send blocked by insufficient buffer space
  - … … … …
- Debugging
  - First find out where each process is blocked at
  - How does it conflict with design? What's wrong with implementation?

Parallel Programming Example: Successive Over Relaxation

- SOR implements a mathematical model for many natural phenomena, e.g., heat dissipation, ocean currents
- Given a 2D grid of data, for some number of iterations:
  - For each internal grid point, compute average of its four neighbors

```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
for( i=1; i<n; i++)
   for( j=1; j<n; j++)
      grid[i][j] = temp[i][j];
```

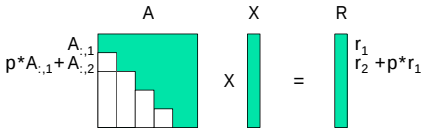

Parallel Programming Example: Successive Over Relaxation

```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
for( i=1; i<n; i++)
   for( j=1; j<n; j++)
      grid[i][j] = temp[i][j];
```

- Task decomposition and data partitioning:
  - 1D partitioning: each process manages some columns or rows
  - 2D partitioning: each process manages a 2D block of the grid
- Message passing:
  - Pass messages on boundary data
  - Send to all neighbors before receiving
  - Nonblocking receive of all neighbors
- 1D or 2D partitioning?

Gaussian Elimination

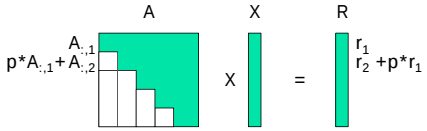
- Reduce an equation matrix into an equivalent upper-diagonal matrix
- Partial pivoting to maintain numerical stability



- Task decomposition and data partitioning:
- Communication/synchronization in row partitioning
  - Find the maximum pivot, distribute the major row
- Communication/synchronization in column partitioning
  - Distribute the major column

Gaussian Elimination

- Reduce an equation matrix into an equivalent upper-diagonal matrix
- Partial pivoting to maintain numerical stability



- Task decomposition and data partitioning:
  - Block vs. cyclic partitioning?

2/16/2011 CSC 258/458 - Spring 2011 13