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
  - More scalable (multiprocessor cache-coherence at large scale is expensive and difficult to build)
- No worry of multiprocessor memory consistency
  - No memory consistency-related errors
- More portable
  - Can run on shared memory multiprocessors
  - Can run on hybrid distributed/shared-memory platforms
- Better locality of parallel programs
  - Programmer knows her/his program best

Parallel Programming Steps
- Converting a sequential application to a parallel one
  - Decompose into tasks
    - Also partition the data
  - Assign tasks to processors
    - Also assign data to processors
  - Orchestrated data access and synchronization
    - Also orchestrate message passing
**Message Passing Interface**

- De facto standard programming interface for message passing-based parallel programs
- 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**

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, …);`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, …);`

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

    ```c
    for (i = 0; i < nprocs; i++)
        if (i != me) {
            MPI_Send(&d, 1, MPI_INT, i, tag, MPI_COMM_WORLD);
        }
    for (i = 1; i < nprocs; i++) {
        MPI_Recv(&d, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
    }
    ```

---

**MPI Send/Receive**

- Alternative send/receive modes:
  - 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, why?
  - carefull data partition and task assignment

- Synchronization/communication wait
  - block as late as possible
  - Receive as late as you can, use nonblocking Receive when you can

- Long communication latency (vs. high bandwidth)
  - our Ethernet cluster: 250us latency, 80MB/sec bandwidth if 1KB per
    synchronization, effective bandwidth 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 get stuck?

Possible reasons
- Group communications are not called by all in the group
- Receive without a matching send
- Matching send for a receive cannot be reached
  - Receive before Send
  - Send before Receive, but 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

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

1D or 2D partitioning?
Nonblocking send/receive as much as possible

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];

Gaussian Elimination

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

A  
X  
R

\[
p^*A_{1,1}+A_{2,2} = \begin{bmatrix} r_1 \\ r_2 + p^*r_3 \end{bmatrix}
\]

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

\[ p^*A_{11} + A_{12} = \begin{bmatrix} \bar{A} \\ \bar{X} \end{bmatrix}, \quad X = \begin{bmatrix} r_1 \\ r_2 + p^*r_1 \end{bmatrix} \]

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

**I/O for Parallel Programs**

- Reading/writing large amount of data from/to storage
  - Parallel processes read a large matrix of data
  - Parallel processes write output statistics
- Using a dedicated I/O process
  - All I/O is issued at the dedicated I/O process; other processes communicate with the I/O process for I/O
  - Problem: scalability

**MPI-I/O**

- Often, each process reads/writes from/to a distinct file/data partition
- MPI-I/O
  - providing data structures to partition data, using standard interface to ease programming (file view, displacement, ...)
  - [http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf](http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf) (Sec 13.3)
  - limitation: not very helpful for accessing irregular data structures (sparse matrices)