L18: MPI, cont.

November 10, 2011

Project 4, Due November 21 at midnight

The code in sparse_matvec.c is a sequential version of a sparse matrix-vector multiply. The matrix is sparse in that many of its elements are zero. Rather than representing all of these zeros which wastes storage, the code uses a representation called Compressed Row Storage (CRS), which only represents the nonzeros with auxiliary data structures to keep track of their location in the full matrix.

I provide:

- Sparse input matrices which were generated from the MatrixMarket (see http://math.nist.gov/MatrixMarket/).
- Sequential code that includes conversion from coordinate matrix to CRS.
- An implementation of dense matvec in CUDA.
- A Makefile for the CADE Linux machines.

You write:

- A CUDA implementation of sparse matvec.

Outline

- Finish MPI discussion
  - Review blocking and non-blocking communication
  - One-sided communication
- Sources for this lecture:
  - http://mpi.deino.net/mpi_functions/
  - Kathy Yelick/Jim Demmel (UC Berkeley): CS 267, Spr 07 • http://www.eecs.berkeley.edu/~yelick/cs267_sp07/lectures

Administrative

- Class cancelled, Tuesday, November 15
- Guest Lecture, Thursday, November 17, Ganesh Gopalakrishnan
- CUDA Project 4, due November 21
  - Available on CADE Linux machines (lab1 and lab3) and Windows machines (lab5 and lab6)
  - You can also use your own Nvidia GPUs
Sparse Linear Algebra

- Suppose you are applying matrix-vector multiply and the matrix has lots of zero elements
  - Computation cost? Space requirements?
- General sparse matrix representation concepts
  - Primarily only represent the nonzero data values
  - Auxiliary data structures describe placement of nonzeros in “dense matrix”

Some common representations

- DIA: Store elements along a set of diagonals.
- Compressed Sparse Row (CSR): Store only nonzero elements, with “ptr” to beginning of each row and “indices” representing column.
- ELL: Store a set of K elements per row and pad as needed. Best suited when number non-zeros roughly consistent across rows.
- COO: Store nonzero elements and their corresponding “coordinates”

Connect to dense linear algebra

Dense matvec from L15:

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        a[i] += c[i][j] * b[j];
    }
}
```

Equivalent CSR matvec:

```c
for (i=0; i<n; i++) {
    for (j = ptr[i]; j<ptr[i+1]-1; j++)
        t[i] += data[j] * b[indices[j]];
}
```

Today's MPI Focus - Communication Primitives

- Collective communication
  - Reductions, Broadcast, Scatter, Gather
- Blocking communication
  - Overhead
  - Deadlock?
- Non-blocking
- One-sided communication
Quick MPI Review

- Six most common MPI Commands (aka, Six Command MPI)
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv

- Send and Receive refer to "point-to-point" communication
- Last time we also showed collective communication
  - Reduce

More difficult p2p example: 2D relaxation

Replaces each interior value by the average of its four nearest neighbors.

Sequential code:

```c
for (j=1; j<n; j++)
  for (i=1; i<n; i++)
    b[i][j] = (a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1])/4.0;
```

MPI code, main loop of 2D SOR computation

```c
111 111 111
111 111 111
111 111 111
```

MPI code, main loop of 2D SOR computation, cont.

```c
111 111 111
111 111 111
111 111 111
```
MPI code, main loop of 2D SOR computation, cont.

```c
average[0]=val[i][j][0]+val[i][j][1]+val[i][j][2]+val[i][j][3]/4;

deltaMaxDelta[0]=abs(average-val[i][j][0])/;

if(deltaMaxDelta[0]<1e-12)
  break;

MPI_Scatter(average, 4, MPI_FLOAT, MPI_MIN, 0, numProcesses, MPI_COMM_WORLD);
```

Broadcast: Collective communication within a group

```c
int main()

int MPI_Scatter()
```

MPI Scatter()

- `MPI_Scatter` function:
  - `void MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int root,```
  - `void *recvbuf, int recvcount, MPI_Datatype recvtype,```
  - `MPI_Comm comm);`  

Arguments:
- The first three arguments specify the address, size, and type of the data elements to scatter to each process. These arguments must have the same data type. If the source process is a root process, then this argument is set to the processor to which the data is to be sent.
- The fourth argument is the rank of the root process to which the data is to be sent.
- The fifth argument is the communicator to use.

Note:
- The `MPI_Scatter` function is used to scatter data from one process to all other processes within a communication group. It takes five arguments: the address of the data to be scattered (`sendbuf`), the number of data elements to scatter (`sendcount`), the type of the data elements (`sendtype`), the root process rank (`root`), and the communicator (`comm`). The function copies the data from the source process to the destination processes.
- The `MPI_Scatter` function returns `MPI_SUCCESS` if the operation is successful, and `MPI_ERROR` if an error occurs. The error code can be checked using the `MPI_Error_string` function.

Distribute Data from input using a scatter operation

```c
int main()
```

- Initialize variables
- Read input data from file
- Use `MPI_Scatter` to distribute data among processes
- Process data on each process
- Print results
Other Basic Features of MPI

- **MPI_Gather**
  - Analogous to **MPI_Scatter**
- **Scans and reductions (reduction last time)**
- **Groups, communicators, tags**
  - Mechanisms for identifying which processes participate in a communication
- **MPI_Bcast**
  - Broadcast to all other processes in a “group”

The Path of a Message

- A blocking send visits 4 address spaces

---

Deadlock?

```c
int a[10], b[10], myrank;
MPI_Status status; ...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
} else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
} ...
```

---

Deadlock?

Consider the following piece of code:

```c
int a[10], b[10], npe, myrank;
MPI_Status status; ...
MPI_Comm_size(MPI_COMM_WORLD, &npe);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npe, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npe)%npe, 1, MPI_COMM_WORLD);
```

Non-Blocking Communication

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a check-status operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non-Blocking Communication

- To overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations ("I" stands for "Immediate"): int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

  These operations return before the operations have been completed.
- Function MPI_Test tests whether or not the non-blocking send or receive operation identified by its request has finished:
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
- MPI_Wait waits for the operation to complete:
  int MPI_Wait(MPI_Request *request, MPI_Status *status)

One-Sided Communication

window

Process 0
Get
Put
Process 1
Process 2
Process 3

= address spaces = = = window object

Improving SOR with Non-Blocking Communication

if (row != Top) {
  MPI_Isend(&val[1][1], Width-2, MPI_FLOAT, NorthPE(myID), tag, MPI_COMM_WORLD, &requests[0]);
} // analogous for South, East and West

if (row != Top) {
  MPI_Irecv(&val[0][1], Width-2, MPI_FLOAT, NorthPE(myID), tag, MPI_COMM_WORLD, &requests[4]);
} // Perform interior computation on local data

// Now wait for Receives to complete
MPI_Waitall(8, requests, status);

// Then, perform computation on boundaries
MPI Constructs supporting One-Sided Communication (RMA)

- MPI_Win_create exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- MPI_Win_free deallocates window object
- MPI_Put moves data from local memory to remote memory
- MPI_Get retrieves data from remote memory into local memory
- MPI_Accumulate updates remote memory using local values

Simple Get/Put Example

```c
i = MPI_Alloc_mem(200 * sizeof(int), MPI_INFO_NULL, &A);
i = MPI_Alloc_mem(200 * sizeof(int), MPI_INFO_NULL, &B);
if (rank == 0) {
    for (i=0; i<200; i++)
        A[i] = B[i] = i;
    MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD,
                   &win);
    MPI_Win_start(group, 0, win);
    for (i=0; i<100; i++)
        MPI_Put(A+i, 1, MPI_INT, 1, i, 1, MPI_INT, win);
    for (i=0; i<100; i++)
        MPI_Get(B+i, 1, MPI_INT, 1, 100+i, 1, MPI_INT, win);
    MPI_Win_complete(win);
    for (i=0; i<100; i++)
        if (B[i] != (-4)*(i+100)) {
            printf("Get Error: B\[i\] is %d, should be %d\n", B[i], (-4)*(i+100));
            fflush(stdout);
            errs++;
        }
} else { /* rank=1 */
    for (i=0; i<200; i++)
        B[i] = (-4)*i;
    MPI_Win_create(B, 200*sizeof(int), sizeof(int), MPI_INFO_NULL,
                   MPI_COMM_WORLD, &win);
    destrank = 0;
    MPI_Group_incl(comm_group, 1, &destrank, &group);
    MPI_Win_post(group, 0, win);
    MPI_Win_wait(win);
    for (i=0; i<100; i++)
        if (B[i] != i) {
            printf("Put Error: B\[i\] is %d, should be %d\n", B[i], i);
            fflush(stdout);
            errs++;
        }
}
```

Simple Put/Get Example, cont.

```c
```

MPI Put and MPI Get

```c
int MPI_Put( void *origin_addr, int origin_count,
           MPI_Datatype origin_datatype, int target_rank,
           MPI_Aint target_disp, int target_count,
           MPI_Win target_datatype, MPI_Win win);
```

```c
int MPI_Get( void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win);
```

Specify address, count, datatype for origin and target, rank for target and MPI_win for 1-sided communication.
**MPI Critique (Snyder)**

- Message passing is a very simple model
- Extremely low level; heavy weight
  - Expense comes from $\lambda$ and lots of local code
  - Communication code is often more than half
  - Tough to make adaptable and flexible
  - Tough to get right and know it
  - Tough to make perform in some (Snyder says most) cases
- Programming model of choice for scalability
- Widespread adoption due to portability, although not completely true in practice

**Summary of Lecture**

- Summary
  - Regular computations are easier to schedule, more amenable to data parallel programming models, easier to program, etc.
  - Performance of irregular computations is heavily dependent on representation of data
  - Choosing this representation may depend on knowledge of the problem, which may only be available at run time