L18: MPI, cont.

November 10, 2011

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

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
  - "Implementing Sparse Matrix-Vector Multiplication on Throughput Oriented Processors," Bell and Garland (Nvidia), SC09, Nov. 2009.

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

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\text{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad \text{offsets} = [-2\ 0\ 1]$$

DIA: Store elements along a set of diagonals.

```
ptr =     [0 2 4 7 9]
indices = [0 1 1 2 0 2 3 1 3]
data =    [1 7 2 8 5 3 9 6 4]
```

Compressed Sparse Row (CSR): Store only nonzero elements, with "ptr" to beginning of each row and "indices" representing column.

$$\text{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad \text{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

ELL: Store a set of K elements per row and pad as needed. Best suited when number non-zeros roughly consistent across rows.

```
row =     [0 0 1 1 2 2 2 3 3]
indices = [0 1 1 2 0 2 3 1 3]
data =    [1 7 2 8 5 3 9 6 4]
```

COO: Store nonzero elements and their corresponding "coordinates".

Connect to dense linear algebra

Dense matvec from L15:

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

Equivalent CSR matvec:

```
for (i=0; i<nr; 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**

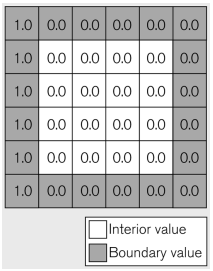
THE UNIVERSITY OF UTAH

More difficult p2p example: 2D relaxation

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

Sequential code:

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



THE UNIVERSITY OF UTAH

MPI code, main loop of 2D SOR computation

```
  1  #define Top      0
  2  #define Left     0
  3  #define Right    (Cols-1)
  4  #define Bottom   (Rows-1)
  5
  6  #define NorthPE(i)       ((i)-Cols)
  7  #define SouthPE(i)       ((i)+Cols)
  8  #define EastPE(i)        ((i)+1)
  9  #define WestPE(i)        ((i)-1)
...
101  do
102  { /*
103    * Send data to four neighbors
104    */
105    if(row !=Top)                              /* Send North */
106    {
107      MPI_Send(&val[1][1], Width-2, MPI_FLOAT,
108               NorthPE(myID), tag, MPI_COMM_WORLD);
109    }
110
111    if(col !=Right)                            /* Send East */
112    {
113      for(i=1; i<Height-1; i++)
114      {
115        buffer[i-1]=val[i][Width-2];
116      }
117      MPI_Send(buffer, Height-2, MPI_FLOAT,
118               EastPE(myID), tag, MPI_COMM_WORLD);
119    }
120
121    if(row !=Bottom)                           /* Send South */
122    {
123      MPI_Send(&val[Height-2][1], Width-2, MPI_FLOAT,
124               SouthPE(myID), tag, MPI_COMM_WORLD);
125    }
126
127    if(col !=Left)                             /* Send West */
128    {
```

THE UNIVERSITY OF UTAH

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

```
129      for(i=1; i<Height-1; i++)
130      {
131        buffer[i-1]=val[i][1];
132      }
133      MPI_Send(buffer, Height-2, MPI_FLOAT,
134               WestPE(myID), tag, MPI_COMM_WORLD);
135    }
136
137    /*
138     * Receive messages
139     */
140    if(row !=Top)                              /* Receive from North */
141    {
142      MPI_Recv(&val[0][1], Width-2, MPI_FLOAT,
143               NorthPE(myID), tag, MPI_COMM_WORLD, &status);
144    }
145
146    if(col !=Right)                            /* Receive from East */
147    {
148      MPI_Recv(&buffer, Height-2, MPI_FLOAT,
149               EastPE(myID), tag, MPI_COMM_WORLD, &status);
150      for(i=1; i<Height-1; i++)
151      {
152        val[i][Width-1]=buffer[i-1];
153      }
154    }
155
156    if(row !=Bottom)                           /* Receive from South */
157    {
158      MPI_Recv(&val[0][Height-1], Width-2, MPI_FLOAT,
159               SouthPE(myID), tag, MPI_COMM_WORLD, &status);
160    }
161
162    if(col !=Left)                             /* Receive from West */
163    {
164      MPI_Recv(&buffer, Height-2, MPI_FLOAT,
165               WestPE(myID), tag, MPI_COMM_WORLD, &status);
166      for(i=1; i<Height-1; i++)
167      {
168        val[i][0]=buffer[i-1];
169      }
170    }
171
172    delta=0.0; /* Calculate average, delta for all points */
173    for(i=1; i<Height-1; i++)
174    {
175      for(j=1; j<Width-1; j++)
176      {
```

THE UNIVERSITY OF UTAH

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

```
177         average=(val[i-1][j]+val[i][j+1]+
178                  val[i+1][j]+val[i][j-1])/4;
179         delta=Max(delta, Abs(average-val[i][j]));
180         new[i][j]=average;
181       }
182     }
183
184     /* Find maximum diff */
185     MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_MIN,
186                RootProcess, MPI_COMM_WORLD);
187     Swap(val, new);
188   } while(globalDelta < THRESHOLD);
```

THE UNIVERSITY OF UTAH

Broadcast: Collective communication within a group

```
1  int numCols;               /* initialized elsewhere */
2
3  void broadcast_example()
4  {
5    int **ranks;             /* the ranks that belong to each group */
6    int myRank;
7    int rowNumber;           /* row number of this process */
8    int random;              /* value that we would like to broadcast */
9    rowNumber=myRank/numCols;
10   MPI_Group globalGroup, newGroup;
11   MPI_Comm rowComm[numCols];
12
13   /* initialize ranks[][] array */
14   ranks[0]={0,1,2,3};  /* not legal C */
15   ranks[1]={4,5,6,7};
16   ranks[2]={8,9,10,11};
17   ranks[3]={12,13,14,15};
18
19   /* Extract the original group handle */
20   MPI_Comm_group(MPI_COMM_WORLD, &globalGroup);
21
22   /* Define the new group */
23   MPI_Group_incl(globalGroup, P/numCols, ranks[rowNumber], &newGroup);
24
25   /* Create new communicator */
26   MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);
27
28   random=rand();
29
30   /* Broadcast 'random' across rows */
31   MPI_Bcast(&random, 1, MPI_, rowNumber*numCols, newComm);
32 }
```

THE UNIVERSITY OF UTAH

MPI_Scatter()

MPI_Scatter()

```
int MPI_Scatter(
  void         *sendbuffer,          // Scatter routine
  int          sendcount,            // Address of the data to send
  MPI_Datatype sendtype,             // Number of data elements to send
  int          destbuffer,           // Type of data elements to send
  int          destcount,            // Address of buffer to receive data
  MPI_Datatype desttype,             // Number of data elements to receive
  int          root,                 // Type of data elements to receive
  MPI_Comm     *comm                 // Rank of the root process
);                                   // An MPI communicator
```

Arguments:

- The first three arguments specify the address, size, and type of the data elements to send to each process. These arguments only have meaning for the root process.
- The second three arguments specify the address, size, and type of the data elements for each receiving process. The size and type of the sending data and the receiving data may differ as a means of converting data types.
- The seventh argument specifies the root process that is the source of the data.
- The eighth argument specifies the MPI communicator to use.

Notes:

This routine distributes data from the root process to all other processes, including the root. A more sophisticated version of the routine, `MPI_Scatterv()`, allows the root process to send different amounts of data to the various processes. Details can be found in the MPI standard.

Return value:

An MPI error code.

THE UNIVERSITY OF UTAH

Distribute Data from input using a scatter operation

```
16  length_per_process=length/size;
17  myArray=(int *) malloc(length_per_process*sizeof(int));
18
19  array=(int *) malloc(length*sizeof(int));
20
21  /* Read the data, distribute it among the various processes */
22  if(myID==RootProcess)
23  {
24    if((fp=fopen(*argv, "r"))==NULL)
25    {
26      printf("fopen failed on %s\n", filename);
27      exit(0);
28    }
29    fscanf(fp,"%d", &length);       /* read input size */
30
31    for(i=0; i<length-1; i++)       /* read entire input file */
32    {
33      fscanf(fp,"%d", myArray+i);
34    }
35  }
36
37  MPI_Scatter(Array, length_per_process, MPI_INT,
38              myArray, length_per_process, MPI_INT,
39              RootProcess, MPI_COMM_WORLD);
```

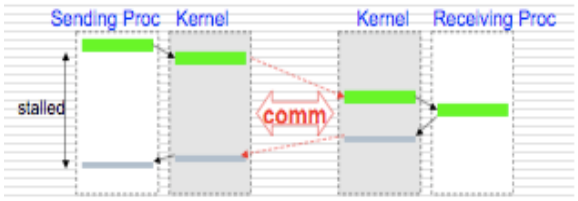
THE UNIVERSITY OF UTAH

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



- Besides being time-consuming, it locks processors together quite tightly

Deadlock?

```
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:

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

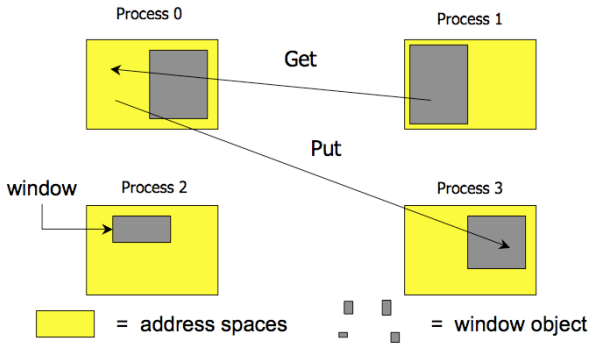
  int MPI_Wait(MPI_Request *request, MPI_Status *status)

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 Recvs to complete
MPI_Waitall(8,requests,status);

//Then, perform computation on boundaries
```

One-Sided Communication



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

MPI_Put and MPI_Get

```
int MPI_Put( 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);

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.

Simple Get/Put Example

```
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++;
    }
}
```

Simple Put/Get Example, cont.

```
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++;
       }
     }
  }
```

MPI Critique (Snyder)

- Message passing is a very simple model
- Extremely low level; heavy weight
  - Expense comes from λ 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