CS4961 Parallel Programming

Lecture 19: Message Passing, cont.

Mary Hall
November 4, 2010

Programming Assignment #3: Simple CUDA Due Thursday, November 18, 11:59 PM

Today we will cover Successive Over Relaxation. Here is the sequential code for the core computation, which we parallelize using CUDA:

```
for(i=1;i<N-1;i++) {
  for(j=1;j<N-1;j++) {
    B[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4;
  }
}
```

You are provided with a CUDA template (sor.cu) that (1) provides the sequential implementation; (2) times the computation; and (3) verifies that its output matches the sequential code.

Programming Assignment #3, cont.

- Your mission:
  - Write parallel CUDA code, including data allocation and copying to/from CPU
  - Measure speedup and report
  - 45 points for correct implementation
  - 5 points for performance
  - Extra credit (10 points): use shared memory and compare performance

Programming Assignment #3, cont.

- You can install CUDA on your own computer
  - http://www.nvidia.com/cudazone/
- How to compile under Linux and MacOS
  - nvcc -I/Developer/CUDA/common/inc \
    -L/Developer/CUDA/lib sor.cu -lcutil
- Turn in
  - Handin cs4961 p03 <file> (includes source file and explanation of results)

Today's Lecture

- More Message Passing, largely for distributed memory
- Message Passing Interface (MPI): a Local View language
- Sources for this lecture
  - Larry Snyder, http://www.cs.washington.edu/education/courses/524/08wi/
  - Online MPI tutorial http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
  - Vivek Sarkar, Rice University, COMP 422, F08 http://www.cs.rice.edu/~vs3/comp422/lecture-notes/index.html
  - http://mpi.deino.net/mpi_functions/

Today's MPI Focus

- 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**

Figure 7.1 An MPI solution to the Count 3s problem.

```
1  #include <stdio.h>
2  #include "mpi.h"
3  #include "globals.h"
4
5  int main(argc, argv)
6  int argc;
7  char **argv;
8  {
9    int myID, value, numProcs;
10   MPI_Status status;
11
12   MPI_Init(&argc, &argv);
13   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
14   MPI_Comm_rank(MPI_COMM_WORLD, &myID);
15
16   length_per_process=length/numProcs;
17   myArray=(int *) malloc(length_per_process*sizeof(int));
18
19   /* Read the data, distribute it among the various processes */
20   if(myID==RootProcess)
21   {
22     if((fp=fopen(*argv, "r"))==NULL )
23     {
24       printf("fopen failed on %s\n", filename);
25       exit(0);
26     }
27     fscanf(fp,"%d", &length);     /* read input size */
28
29     for(p=0; p<numProcs-1; p++)   /* read data on behalf of each */
30     {                             /* of the other processes */
31       for(i=0; i<length_per_process; i++)
32       {
33         fscanf(fp,"%d", myArray+i);
34       }
35       MPI_Send(myArray, length_per_process, MPI_INT, p+1,
36                tag, MPI_COMM_WORLD);
```

Figure 7.1 An MPI solution to the Count 3s problem. *(cont.)*

```
37      }
38
39      for(i=0; i<length_per_process; i++)  /* Now read my data */
40      {
41        fscanf(fp,"%d", myArray+i);
42      }
43    }
44    else
45    {
46      MPI_Recv(myArray, length_per_process, MPI_INT, RootProcess,
47               tag, MPI_COMM_WORLD, &status);
48    }
49
50    /* Do the actual work */
51    for(i=0; i<length_per_process; i++)
52    {
53      if(myArray[i]==3)
54      {
55        myCount++;                          /* Update local count */
56      }
57    }
58
59    MPI_Reduce(&myCount,&globalCount, 1, MPI_INT, MPI_SUM,
60               RootProcess, MPI_COMM_WORLD);
61
62    if(myID==RootProcess)
63    {
64      printf("Number of 3's: %d\n", globalCount);
65    }
66    MPI_Finalize()
67    return 0;
68  }
```


Code Spec 7.8 `MPI_Scatter()`.

`MPI_Scatter()`

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

(continued)

Code Spec 7.8 `MPI_Scatter()`. *(cont.)*

- 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.

Figure 7.2

Replacement code (for lines 16-48 of Figure 7.1) to distribute data 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);
```

Other Basic Features of MPI

- MPI_Gather
  - Analogous to MPI_Scatter
- Scans and reductions
- Groups, communicators, tags
  - Mechanisms for identifying which processes participate in a communication
- MPI_Bcast
  - Broadcast to all other processes in a "group"

Figure 7.4 Example of 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 }
```


Figure 7.5 A 2D relaxation replaces—on each iteration—all interior values by the average of their 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;
```

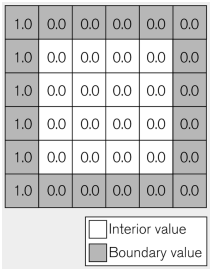


Figure 7.6 MPI code for the main loop of the 2D SOR computation.

```
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   {
```

Figure 7.6 MPI code for the main loop of the 2D SOR computation. *(cont.)*

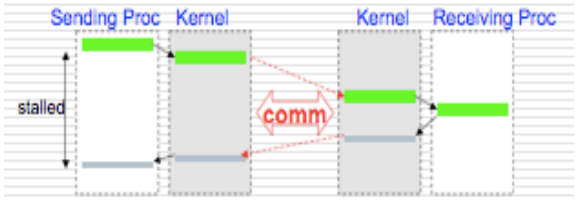
```
129     for(i=1; i<Height-1; i++)
130     {
131       buffer[i-1]=val[i][1];
132     }
133     MPI_Send(buffer, Height-2, MPI_FLOAT,
134              WestPE(myID), tag, MPI_COMM_WORLD);
135   }
136
137   /*
138    * Receive messages
139    */
140   if(row !=Top)                         /* Receive 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     {
```


Figure 7.6 MPI code for the main loop of the 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 Path of a Message

- A blocking send visits 4 address spaces



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

Non-Buffered vs. Buffered Sends

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling at the expense of copying overheads.

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.

Flavors of send, p. 214 in text

- Synchronous send (MPI_Ssend()):
  - Sender does not return until receiving process has begun to receive its message
- Buffered Send (MPI_Bsend()):
  - Programmer supplies buffer for data in user space, useful for very large or numerous messages.
- Ready Send (MPI_Rsend()):
  - Risky operation that sends a message directly into a memory location on the receiving end, and assumes receive has already been initiated.

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

- 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.

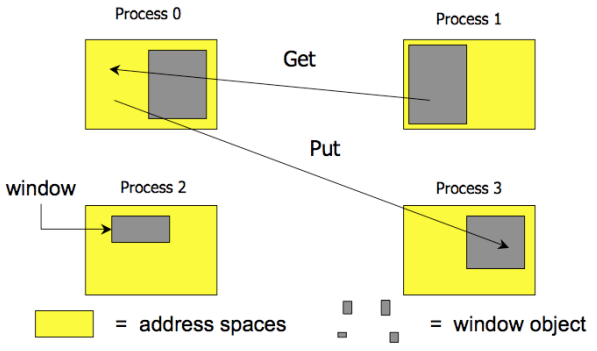
  int MPI_Wait(MPI_Request *request, MPI_Status *status)

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 One-Sided Communication or Remote Memory Access (RMA)

- Goals of MPI-2 RMA Design
  - Balancing efficiency and portability across a wide class of architectures
  - shared-memory multiprocessors
  - NUMA architectures
  - distributed-memory MPP's, clusters
  - Workstation networks
- Retaining "look and feel" of MPI-1
- Dealing with subtle memory behavior issues: cache coherence, sequential consistency

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

```
i = MPI_Alloc_mem(SIZE2 * sizeof(int), MPI_INFO_NULL, &A);
i = MPI_Alloc_mem(SIZE2 * 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, commgrp, &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 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++;
    }
}
```


Get/Put 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<SIZE1; 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