L17: MPI, cont.

October 25, 2012

Final Project

- Purpose:
  - A chance to dig in deeper into a parallel programming model and explore concepts.
  - Research experience
    - Freedom to pick your own problem and solution, get feedback
    - Thrill of victory, agony of defeat
  - Communication skills
    - Present results to communicate technical ideas
- Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations.
  - OpenMP + SSE
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - MPI + OpenMP
  - MPI + SSE
  - MPI + CUDA
- Present results in a poster session on the last day of class

Example Projects

- Look in the textbook or look on-line
  - Chapter 6: N-body, Tree search
  - Chapters 3 and 5: Sorting
  - Image and signal processing algorithms
  - Graphics algorithms
  - Stencil computations
  - FFT
  - Graph algorithms
  - Other domains…
- Must change it up in some way from text
  - Different language/strategy

Details and Schedule

- 2-3 person projects
  - Let me know if you need help finding a team
- Ok to combine with project for other class, but expectations will be higher and professors will discuss
- Each group must talk to me about their project between now and Thanksgiving
  - Before/after class, during office hours or by appointment
  - Bring written description of project, a few slides are fine
  - Must include your plan and how the work will be shared across the team
- I must sign off by November 20 (in writing)
- Dry run on December 4
- Poster presentation on December 6

Strategy

- A lesson in research
  - Big vision is great, but make sure you have an evolutionary plan where success comes in stages
    - Sometimes even the best students get too ambitious and struggle
  - Parallel programming is hard
    - Some of you will pick problems that don't speed up well and we'll need to figure out what to do
  - There are many opportunities to recover if you have problems
    - I'll check in with you a few times and redirect if needed
    - Feel free to ask for help
    - Optional final report can boost your grade, particularly if things are not working on the last day of classes

Outline

- Finish MPI discussion
  - Review blocking and non-blocking communication
  - Introduce one-sided communication
- Sources for this lecture:
  - http://mpi.deino.net/mpi_functions/

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**

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:

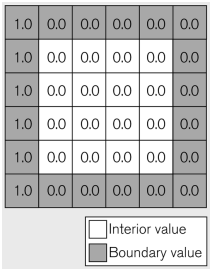
```
int a[10], b[10], npes, myrank;
MPI_Status status; ...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
  MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
  MPI_COMM_WORLD); ...
```


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;
```



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    {
```

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++)
```


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);
```


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.

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);
```

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.

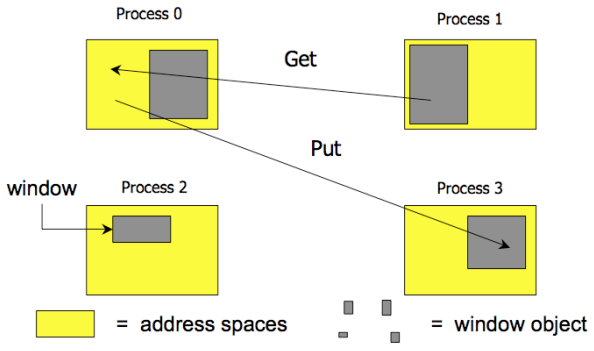
  int MPI_Wait(MPI_Request *request, MPI_Status *status)

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.

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