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

```c
for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
    }
}
```

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

- **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 -filer (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
  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

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  int size, rank, myrank, msg; int* tree; int* len2; int length; int* 3s; int* 2s;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0)
  {
    tree = (int*) malloc(size * sizeof(int));
    len2 = (int*) malloc(size * sizeof(int));
    length = (int*) malloc(size * sizeof(int));
    3s = (int*) malloc(size * sizeof(int));
    2s = (int*) malloc(size * sizeof(int));
    for (int i = 0; i < size; i++)
    {
      // initialize tree, len2, length, 3s, 2s
      // read input
      // sort
    }
    for (int i = 0; i < size; i++)
    {
      // print output
    }
    MPI_Finalize();
  } else
  {
    // receive
    // process
    // send
    // free memory
  }
  return 0;
}
```
Figure 7.1  An MPI solution to the Count 3s problem. (cont.)

Code Spec 7.8  MPI Scatter().

```
 mpi_scatter();
```

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 result of converting data types.

Notes: This function 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.

Figure 7.2 Replacement code (for lines 16–48 of Figure 7.1) to distribute data using a scatter operation.

```c

```

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.
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"

---

Sequential code:

```c
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 7.5  A 2D relaxation replaces—on each iteration—all interior values by the average of their four nearest neighbors.

---

Figure 7.4  Example of collective communication within a group.

```c
1. let numprocs: /* number of processes */
2. let rworld_rank: /* rank on this machine */
3. let rworld_size: /* total number of processes on all the machines */
4. let root: /* the root, if it exists */
5. let offset: /* the offset to the root process */
6. let local_rank: /* the rank of a process on this machine */
7. let local_size: /* the number of processes on this machine */
8. let comm: /* the communicator */
9. let my_rank: /* the rank of the current process */
10. let my_size: /* the number of processes on this machine */
11. if (my_rank == root) {
12.     /* broadcast the root's data */
13.     MPI_Bcast(data, size, datatype, root, comm);
14.   }
15. else { /* do nothing */
16.   }
```

---

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

```c
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 Path of a Message

- A blocking send visits 4 address spaces

![Diagram showing the path of a message through 4 address spaces: Sending Proc, Kernel, Kernel, Receiving Proc.]

- 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 the 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?

```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], 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**

```c
int i;
int A[200];
int B[200];
int group[200];

int main(int argc, char **argv) {
    int rank, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* For simplicity, assume each process has the same number of elements */
    int size = 200;
    int size2 = size * sizeof(int);

    i = MPI_Alloc_mem(size2, MPI_INFO_NULL, &A);
    i = MPI_Alloc_mem(size2, MPI_INFO_NULL, &B);

    if (rank == 0) {
        for (i = 0; i < size; 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; i++)
            MPI_Get(B + i, 1, MPI_INT, 1, 100 + i, 1, MPI_INT, win);
        MPI_Win_complete(win);
    }

    else { /* rank=1 */
        for (i = 0; i < size; 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 < size; i++)
            if (B[i] != i)
                printf("Put Error: B[%d] is %d, should be %d\n", B[i], i);
                fflush(stdout);
                errs++;
    }

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

**Get/Put Example, cont.**

```c
else { /* ranks!=1 */
    for (i = 0; i < size; 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_wait(win);
    for (i = 0; i < size; i++)
        if (B[i] != i)
            printf("Put Error: B[%d] is %d, should be %d\n", B[i], i);
            fflush(stdout); errrs++;

    } else { /* rank=1 */
```

**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