CSCI 503 Homework 3

-Prasanth S Iyer
(piyer@usc.edu)
1958-5902-92

1. **Please discuss the advantages the message passing programming model has over the shared memory programming model; and disadvantages (if any) [3 points].**

   Advantages of the message passing model over the shared memory model:

   1) In case of the message passing model, each node has access only to local data and any data that needs to be shared has to be explicitly passed between nodes. Consequently, synchronization of memory access and computations between the nodes is much simpler. However, in case of the shared memory model, memory accesses and computations on shared data need to be explicitly synchronized.

   2) Maintaining restricted access of data is much simpler in the message passing model since each node has access only to a particular subset of the data. In case of the shared memory model, access to data would have to be explicitly restricted.

   3) The message passing model is much more scalable over large numbers of nodes sharing a communication network where maintaining a shared memory architecture is much more expensive and complicated. Also, in cases where only a small subset of data is required by each node, the message passing model is much more efficient.

   Disadvantages of the message passing model over the shared memory model:

   1) The shared memory model is usually much simpler to program because there does not need to be any explicit splitting and communication of data between the nodes as in the case of the message passing model. This is especially true in cases where the distribution of data is not uniform among the nodes as message passing would require specialized calculations for each node to split the data among them.

   2) The shared memory model is also more efficient where all nodes need to perform computations on a large portion of the data since all the data is implicitly shared between the nodes. In this case, the message passing model would have to explicitly send all the data to all nodes which may take up a lot of time. This is also true when large numbers of extremely small non-contiguous portions of data need to be shared since the overhead in sending the relatively small portions of data to all nodes would be extremely high.

   3) The shared memory model is much more plausible in cases where the architecture implicitly supports shared memory - for instance, in case of multiple processors sharing the same memory. In this case, since the memory is already implicitly shared, passing messages between the processors would just be wasteful.

2. **Based on your programming assignment, please discuss what is your understanding of a compute kernel [3 points].**

   In the context of the mean and median filters used in the programming assignment, my understanding of a compute kernel is that it can be defined for each pixel in the image as the set of all pixels involved in computing the value for that pixel in the image.

   By this definition, in case of both the mean and median filters, for each pixel, the compute window consists of a windowSize x windowSize matrix centered at that pixel. A graphical representation of the compute kernel would be as follows:-

   Current pixel = (x,y)
   Ws = WindowSize

(x-Ws, y-Ws)				(x, y-Ws)				(x+Ws, y-Ws)
				...				
				(x,y-2)				
			...	(x,y-1)	...			
(x-Ws, y)	...	(x-2,y)	(x-1,y)	(x,y)	(x+1,y)	(x+2,y)	...	(x+Ws, y)
			...	(x, y+1)	...			
				(x,y+2)				
				...				
(x-Ws, y+Ws)				(x,y+Ws)				(x+Ws, y+Ws)

Note that this would be slightly modified in case of the edge-conditions discussed in the next question.

In case of the mean filter, the value of the pixel in the center of the compute window would be replaced by the mean of all the values in the compute window whereas in case of the median filter, the value of the pixel in the center of the compute window would be replaced by the median of all the values in the compute window.

3. Based on your programming assignment, please discuss the edge-condition [3 points].

In my solution, I divided the work among the available processors in the following manner:-

Terminology:
Number of rows in the image: M
Number of columns in the image: N
Number of available processors: P
Window size used for the filter: WindowSize

1) The manager process divides the MxN image into P rectangular chunks each of size $M/\sqrt{P}$ x $N/\sqrt{P}$. These are the chunks for which the filters need to be calculated by each worker process. However, in order to do that, each worker needs additional rows and columns based on the WindowSize.

2) So, the manager process sends each worker process a rectangular chunk of size ($M/\sqrt{P}$ + WindowSize) x ($N/\sqrt{P}$ + WindowSize). The additional WindowSize is calculated in the following manner:-

    a. Assume that a worker process is to calculate the filter for a chunk bordered by
    Left top: (startrow, startcol)
    Right bottom: (startrow + $M/\sqrt{P}$, startcol + $N/\sqrt{P}$)

    b. The master process then sends this worker process a chunk bordered by
    Left top: (startrow - WindowSize/2, startcol - WindowSize/2)
    Right bottom: (startrow + $M/\sqrt{P}$ + WindowSize/2, startcol + $N/\sqrt{P}$ + WindowSize/2)

3) I assume that the worker process is to calculate the filter for a chunk bordered by:
Left top: (startrow, startcol)
Right bottom: (startrow + $M/\sqrt{P}$, startcol + $N/\sqrt{P}$)

I also assume that the manager uses 4 variables startx, starty, endx and endy to create the chunk to be sent to the worker. Initially,
startx = startrow - WindowSize/2
starty = startcol - WindowSize/2
endx = startrow + $M/\sqrt{P}$ + WindowSize/2
endy = startcol + $N/\sqrt{P}$ + WindowSize/2

With these assumptions, I encountered 4 different edge-conditions:-

a. *startrow = 0 (The chunk includes the first row of the image)*
   In this case, the master sets startx = 0.

b. *startcol = 0 (The chunk includes the first column of the image)*
   In this case, the master sets starty = 0.

c. *startrow + M/√P + WindowSize/2 ≥ M (The chunk includes the last row of the image)*
   In this case, the master sets endx = M.

d. *startcol + N/√P + WindowSize/2 ≥ N (The chunk includes the last column of the image)*
   In this case, the master sets endy = N.

The master process then sends this worker process a chunk bordered by:
Left top: (startx, starty)
Right bottom: (endx, endy)

Note that this also handles all combinations of these edge conditions as well. This was the approach I used to handle all edge-conditions.

4. **You are not required to comprehensive timing analysis for the programming assignment. However, please discuss how/why a median filter might stand to gain more for the parallel implementation than a mean filter [5 points].**

- The main difference between a mean filter and a median filter is the amount of computation involved in each of them:-

1) In case of the mean filter, the amount of computation involved is restricted to simply summing all the elements of the compute window for each pixel. Consequently, the run time is O(MxNxk) where

   M = Number of rows in the image
   N = Number of columns in the image
   k = Size of the compute window = WindowSize x WindowSize

2) On the other hand, in case of the median filter, the amount of computation involved also consists of sorting all the elements in the compute kernel. Consequently, the run time as it is implemented in the assignment is O(MxNx(k + k logk)) assuming sort(...) uses an O(nlogn) sorting algorithm.

- According to Amdahl's Law, the overall gain by parallelization is
  $S(P)=1/[(1-A)+A/P]$
  Where A is the portion of the program that can be parallelized and P is the number of processors.

- In case of the mean filter, A consists of simply summing up all the elements of the compute window for each pixel whereas, in case of the median filter, A also includes sorting the elements of the compute window for each pixel. Based on the complexity analysis mentioned above, for the same image size and the same window size, $A_{median} > A_{mean}$

- Thus, the portion of the program that can be parallelized is more in case of the median filter as compared to the mean filter given that the image size and the window size are the same.

- Consequently, based on Amdahl's law, the median filter would stand to gain more for the same image and window sizes compared to a mean filter for the same number of nodes P.

5. **Given the unsafe MPI programming practice below, please describe the problem with the code and if applicable please recommend a solution [10 points].**

```
...
if (rank == 0) {
 for(i=0; i<4096; i++)
    data[i] =  'x';

 start = MPI_Wtime();
 while (1) {
  MPI_Send(data, 4096, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
  count++;
  if (count % 100 == 0) {
    end = MPI_Wtime();
    printf("Count = %d  Time= %f sec.\n", count, end-start);
    start = MPI_Wtime();
    }
   }
 }
if (rank == 1) {
 result = 0.0;
 while (1) {
  MPI_Recv(data, 4096, MPI_BYTE, source, tag, MPI_COMM_WORLD,
    &status);
  for (i=0; i < 1000000; i++)
    result = result + (double)random();
   }
}
....
```

Solution:

- The main problem with the above code is that for each iteration of the infinite loop, the receiver has more processing to do than the sender. The reason why this is a problem is that the message passing protocol to be used by MPI_Send is not defined by the MPI standard. Consequently, there are 2 ways of implementing MPI_Send:-

  - **Eager -** An asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive. In this case, it is assumed that the receiver has enough system buffer space to buffer all incoming messages and MPI_Send does not wait for a corresponding MPI_Receive.

  - **Rendezvous** - A synchronous protocol which requires an acknowledgement from a matching receive in order for the send operation to complete. In this case, MPI_Send would only return after it encounters a corresponding MPI_Receive.

    Some implementations may even use a combination of the two using eager protocol for short messages and rendezvous for long messages.

- Consequently, there is a possibility that if the MPI implementation is using an eager protocol, MPI_Send would return successfully even though a corresponding

MPI_Receive has not been encountered. The sent messages have to be stored in the receiver's system buffer.

- As MPI_Send is called in an infinite loop and since the receiver has more processing to do than the sender, there is a possibility that while using eager mode, the sender sends so many messages to the receiver that its system buffer becomes full. This is because in eager mode, before the receiver can acknowledge the messages using MPI_Receive and store them in a corresponding memory buffer, the messages are stored in the system buffer.

- Since the successful execution of the program depends on the implementation of MPI_Send and the size of the system buffer, this program is unsafe.

- Possible solution:
  - A very simple fix would be to use MPI_Ssend instead of MPI_Send. Based on the MPI standard, MPI_Ssend will not return until matching receive is posted. Consequently, this would ensure that as long as the system buffer has space for 4096 bytes, the MPI_Ssend would only return after MPI_Receive has been posted.