Name: Sanju Kurubara Budi Hall Hiriyanna Gowda

Student Id: 800953525

Assignment 6

1. Fill the following table. For each algorithm and each network structure, answer the following questions. Run a small example if you have difficulty seeing how communication happens; but express all answers for the case with P processors.

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Most loaded link | Most loaded node | Longest chain of communication |
| Reduce-star on chain | p1->p0(most loaded link)  θ(p) | p0(most loaded node)  Load = (n/p + p)/ (n/p) | Longest = P  pp-1 -> p0  θ(p) |
| Reduce-star on clique | Many direct links exists  θ(1) | p0(most loaded node)  Load = (n/p + p)/ (n/p) | Longest = 1  pn -> p0  θ(1) |
| Reduce-chain on chain | Pn->p0(All are equally loaded)  θ(1) | All nodes have same load  Load = n/p | Longest = 1  pn -> pn-1  θ(1) |
| Reduce-chain on clique | Pn->p0(all are equally loaded)  θ(1) | All nodes have same load  Load = n/p | Longest = P  Pp-1 -> p0  θ(p) |
| Reduce-tree on chain | p0->p1 (most loaded link)  θ(p) | Comp(p) + p\*comm(1) | Longest = log(P)  Pp-1 -> p0  θ(logP) |
| Reduce-tree on clique | Pn->p0 (all are equally loaded)  θ(p) | Comp(log p) + p\*comm(n/p) | Longest = log(P)  Pp-1 -> p0  θ(logP + 1) |

What do you think is the best algorithm for each network structure? (One of the given algorithm or a different one.)

**Chain network**

Reduce-chain on chain seems to be the best option for chain network with less link load and less communication compared to Reduce-star or Reduce-tree

**Clique network**

Reduce-tree on clique seems to be the best option for clique network with only log p computations compared to Reduce-star and Reduce-chain

1. For each partitioning, write the algorithm that computes heat equation using this decomposition.

**Algorithm for Round robin partition**

|  |
| --- |
| HeatRoundRobin(Heat, p, P, N)  {  // P blocks for 1st round. Hence there are N/P rounds  //creating local array which stores all the elements for the current process p  for (int i = 0 ; i < P ; i++)  {  local[i] = ((N/P) \* i) + p  }    int j = 0;  while (j<P)  {  if(local[j] == 0)  {  Heat[0] = (2 \* Heat[0] + Heat[1])/3  send(Heat[0]) to process 1  }  else if (local[j] != N-1) //until N-1, excluding N-1  {  recieve(Heat[local[j]-1]) from p-1  recieve(Heat[local[j]+1]) from p+1  Heat[local[j]] = (Heat[local[j]-1] + Heat[local[j]] + Heat[local[j]+1])/3  send(Heat[local[j]]) to p+1  }  else // for N-1  {  recieve(Heat[N-2]) from p-2  Heat[N-1] = (2 \* Heat[N-1] + Heat[N-2])/3  }  j++;  }  } |

**Algorithm for Block partition**

|  |
| --- |
| HeatBlock(Heat, p, P, N)  {  // start and end defines start and end of pth block  start = p \* (N/P)  end = ((p+1) \* (N/P)) - 1    for (int i = start; i <= end;i++)  {  if (i == 0)  {  Heat[0] = (2\*Heat[0] + Heat[1])/3  }  else if (i != (N - 1))  {  if (i == start)  recieve(prevVal) from p-1  else  prevVal = Heat[i-1]    Heat[i] = (Heat[i] + prevVal + Heat[start + 1])/3    if (i == end)  {  send(Heat[end]) to p+1  }  }  else //i == (N-1)  {  Heat[N-1] = (2\*Heat[N-1] + Heat[N-2])/3  }  }  } |

For each partitioning, how much communication happen per iteration of the heat equation?

**Round robin partition**

Communication per iteration = θ(P)

**Block partition**

Communication per iteration = θ(1)

What data partitioning would you use?

Communication is very less in Block partition. Total there are only P communications between nodes. However, in Round robin partition, there are total N communications between nodes. So, it’s always better to choose **Block partition** over Round robin partition.

1. Write the algorithm that performs y = Ax; x = y; 10 times in a loop.

**Dense matrix horizontal**

|  |
| --- |
| // A -> Matrix, x -> array, p->processor id, P ->total processors, N->matrix and array size  DenseMatHorizontal(A, x, p, P, N)  {  for (int iter = 0; iter<10; iter++)  {  start = p\*(N/P)  end = ((p + 1) \* ( N/P )) - 1  Create localSum array of size N.  for (int i = start; i <= end; i++)  {  for (int j = 0; j < N; j++)  {  localSum[j] = localSum[j] + (A[i][j] \* x[j])  }  }  if (p != 0)  {  send(localSum) to process 0  }  else  {  // to generate final y values after receiving computed sum array from other processes    //copying localSum array from process 0 to y  for (i = start; i<= end; i++)  y[i] = localSum[i]    for (int i=1; i < P; i++)  {  recieve(localSum) array from process i  copy received localSum array to y.  }  }  }    } |

**Dense matrix vertical**

|  |
| --- |
| DenseMatVertical(A, x, p, P, N):  {  for (int iter = 0; iter < 10; iter++)  {  start = p \* (N/P)  end = ((p + 1) \* (N/P)) - 1    if (iter == 0)  create and initialize y array    Create localSum array of size end-start.  for (int i = 0; i < N; i++)  {  for (int j = start; j <= end; j++)  {  localSum[i] = localSum[i] + (A[i][j] \* x[j])  }  }    if (p == 0)  {  // to generate final y values after receiving computed sum array from other processes    //copying localSum array from process 0 to y  for (i = start; i<= end; i++)  y[i] = localSum[i]    for (int i=1; i < P; i++)  {  recieve(localSum) array from process i  copy received localSum array to y.  }  }  else  {  send(localSum) to process 0  }    }  } |

**Dense matrix block**

|  |
| --- |
| DenseMatBlock(A, x, p, P, N):  {  for (int iter = 0; iter < 10; iter++)  {  start = p \* (N/sqrt(P))  end = ((p + 1) \* (N/sqrt(P))) - 1    if (iter == 0)  create and initialize y array    Create localSum array of size end-start.  for (int i = start; i <= end; i++)  {  for (int j = start; j <= end; j++)  {  localSum[i] = localSum[i] + (A[i][j] \* x[j])  }  }    if (p == 0)  {  // to generate final y values after receiving computed sum array from other processes    //copying localSum array from process 0 to y  for (i = start; i<= end; i++)  y[i] = localSum[i]    for (int i=1; i < P; i++)  {  recieve(localSum) array from process i  copy received localSum array to y.  }  }  else  {  send(localSum) to process 0  }    }  } |

How much memory does each node need?

**Horizontal data partition**

All p process creates array of size N. Also process 0 creates array of size N to collect from other processes and store them in final array.

Hence, memory = θ(N\*P + N)

**Vertical data partition**

Here same created array can be used in all the processes. Process 0 creates an array of size N. A pointer to update will be sent to other processes.

Hence, memory = θ(N)

**Block data partition**

Reason is same as Vertical data partition.

Memory = θ(N)

How much communication does the algorithm do per iteration?

**Horizontal data partition**

All p process creates array of size N and it will be sent to next process only once. Hence total communication per iteration = θ(P)

**Vertical data partition**

Even though partition is different total computed array will be sent only once to next process. Hence, communication per iteration = θ(P)

**Block data partition**

Reason is same as Vertical data partition.

Communication per iteration = θ(P)