ITCS 5154

PARALLEL COMPUTING

Rahul Rachapalli
800968032

Table of Contents

Disclaimer:

This document is a copy of *README.md* file from one of my repositories. The original can be found at https://github.com/rahulr56/OpenMP/tree/master/DistributedComputing

Reduction

Case	Most loaded link	Most loaded node	Longest chain of communication
Reduce-star on Chain	Link between nodes 0 & 1. $\theta(P)$	Node 0. $\theta(\text{Computation}(\frac{N}{P}) + \text{Communication}(P))$	Node (P-1) to Node 0. $\theta(P\ \text{Communication}(1))$ on Node 0
Reduce-star on Clique	All nodes send message to Node 0. 0 or $\theta(1)$	Node 0 $\theta(\text{Computation}(\frac{N}{P}) + \text{Communication}(P))$	All nodes communicate to Node 0. $\theta(1)$
Reduce-chain on Chain	All the same $\theta(1)$	All the same $\theta(\text{Computation}(N/P))$	Every node communicates with the next node. ie., P+1 with P $\theta(1)$
Reduce-chain on Clique	All the same $\theta(1)$	All the same $\theta(\text{Computation}(N/P))$	Node (P-1) to Node 0 $\theta(P)$
Reduce-tree on chain	Link between Nodes 0 & 1. $\theta(P)$	Node 0 $\theta(\text{Computation}(P) + P\ \text{Communication}(1))$	Node (P-1) to Node 0 → $\theta(P)$ Case when a message from P to P-2 includes P-1. $\theta(\log(P))$ when P-1 is excluded from comm. between P & P-2.
Reduce-tree on clique	All the same $\theta(1)$	Node 0 $\theta(\text{Communication}(\log P) + \text{Computation}(N/P))$	Node (P-1) to Node 0 $\theta(\log(P) + 1)$

Best Algorithm

- Chain Network:

  Reduced-Chain gives the best performance for network load, minimal communication over Reduced-star and Reduced-tree. Hence, it is preferred for Chain network structure.

- Clique Network:
  Reduced tree give the best performance with $\theta(\log(P)+1)$ communications over $\theta(P)$communications in reduced-clique. However, the node 0 is a bit more loaded ie., $\theta(\text{Communication}(\log(P)) + \text{Computation}(N/P))$ than other nodes with $\theta(\text{Computation}(N/P))$.

Heat Equation

Algorithm for Block data partition

```
calculateHeatUsingBlock(heatArr, N , p, P)
{
    begin = p * (N/P)
    end = (p+1) * (N/P)
    if (begin == 0)
    {
        heatArr[0] = (2*heatArr[0] + heatArr[1])/3
    }
    else
    {
        recv heatVal from p-1
        heatArr[begin] = (heatArr[begin] + heatVal + heatArr[begin + 1])/3
    }
    if (end == N)
    {
        heatArr[N-1] = (2*heatArr[N-1] + heatArr[N-2])/3
        end = N-1
    }
    for ( i = begin+1 ;  i < end ; ++i)
    {

        heatArr[begin] = (heatArr[begin - 1] + heatArr[begin] + heatArr[begin + 1])/3
    }
    if (p != p-1)
    {
        send heatArr[end-1] to p+1
    }
}
```

Communication per iteration is

```
θ(1) or 0
```

Total Communication:

```
θ( P * Communication(1) )
```

Algorithm for Round Robin data partition

```
calculateHeatUsingRoundRobin(heatArr, N, p, P)
{
    create an array X of size P
    for (i = 0 ; i < P ; ++i)
    {
        X[i] = (N/P) * i + p
    }

    for (i = 0; i < P; ++i)
    {
        if(X[i] == 0)
        {
            heatArr[0] = (2 * heatArr[0] + heatArr[1])/3
            send heatArr[0] to p1
        }
        else if (X[i] == N-1)
        {
            recv heatArr[N-1] from P-1
            heatArr[N-1] = (2 * heatArr[N-1] + heatArr[N-2])/3
        }
        else
        {
            recv heatArr[X[i]-1] from P-1
            heatArr[X[i]] = (heatArr[X[i]-1] + heatArr[X[i]] + heatArr[X[i+1]])/3
            send heatArr[X[i]] to p+1
        }
    }
}
```

Total Communication:

```
θ( N * Communication(1) )
```

Communication per iteration is:

```
θ( P * Communication(1) )
```

Best Algorithm

Since the communication is very costly in a distributed computing environment, I would choose block data partitioning over Round Robin data partition. Suppose that the cost of communication is less than that of the wait time, I would prefer Round Robin data partitioning over block data partition among the above-mentioned data partition algorithms.

I personally would implement ***"All gather operation"*** on the data which incurs a communication over head at the beginning but, reduces the wait time taken. This is particularly useful when the communication cost is cheaper than wait time costs.

Dense Matrix Multiplication

Algorithm for Horizontal data partition(Star)

```
matmulHorizontalPartitioning(A, vectorX, N, p, P)
{
    for (k = 0; k < 10; ++k)
    {
        if(p==0)
        {
            create an array Y of size N and initialize array elements to 0.
        }
        create an array computedVal of size (N) and initialize array elements to 0
        begin = p * (N/P)
        end = (p + 1) * (N/P)
        for (i = begin; i < end; ++i)
        {
            for (j = 0; j < N; ++j )
            {
                computedVal[i] += (A[i][j] * vectorX[j])
            }
        }
        if (p == 0)
        {
            copy computedVal array of Node 0 to Y.
            for (r = 1; r < P; ++r)
            {
                recv computedVal from node r
                Y += computedVal[r]
            }
            vectorX = Y
        }
        else
        {
            send computedVal to Node 0
        }
    }
}
```

Memory Consumed: Every node other than Node 0 creates an array of size `N`. Node 0 creates 2 arrays one to compute values and the other to store the end result. So, their total memory consumption is `θ(P*N+N) = *θ((P+1)*N)`.

Communication per iteration: Each node computes the value of a 1d array of size `N`. The algorithm requires `θ(P)`communications per iteration to update the vector X and redo the whole matrix multiplication with updated vector X. The exact number of communications per iteration is:

```
θ( P * Communication(N) )
```

`Communication(N)` implies the communication overhead required to send an array of size `N`.

Algorithm for Vertical data partition (Chain)

```
matmulVerticalPartitioning(arr, rows, cols, p, P):
{
    for ( k = 0; k < 10; ++k)
    {
        begin = p * (N/P)
        end = (p + 1) * (N/P)
        if(p != 0)
        {
            recv computedVal from p-1
        }
        else if (p == 0  && k == 0)
        {
            if(k != 0)
            {
                recv computedVal from P-1
                vectorX = computedVal
                reinitilize all vallues in computedVal to 0s.
            }
            else
            {
                create and initialize an array computedVal of size N to 0
            }
        }
        for (i = 0; i < N; ++i)
        {
            for(j = begin; j < end; ++j)
            {
                computedVal[i] += (arr[i][j] * vectorX[j])
            }
        }
        if(p != P-1)
        {
            send computedVal to p+1
        }
        else if(p == P-1  && k == 9)
        {
            print computedVal array
        }
        else if(p == P-1)
        {
            send computedVal to 0
        }
    }
}
```

Memory Consumed: Node 0 creates an array of size `N` to store the computed results and sends this array to the adjacent node. The next node computes values and stores them in the same array. Hence, the total memory consumption is `θ(N)`

Communication per iteration: As this algorithm is CHAIN structured, there are `θ(Communication(N) * P)` communications happening in every iteration. `Communication(N)` implies the communication overhead required to send an array of size `N`.

Algorithm for Block data partition (Chain)

```
matmulBlockPartitioning(A, vectorX, N, p, P)
{
    for ( k = 0; k < 10; ++k)
    {
        begin = p * (N/P)
        end = (p + 1) * (N/P)
        if(p != 0)
        {
            recv computedVal from p-1
        }
        else if (p == 0  && k == 0)
        {
            if(k != 0)
            {
                recv computedVal from P-1
                vectorX = computedVal
                reinitialize computedVal to 0s
            }
            else
            {
                create and initialize an array computedVal of size N to 0
            }
        }
        for (i = begin; i < end; ++i)
        {
            for(j = begin; j < end; ++j)
            {
                computedVal[i] += (arr[i][j] * vectorX[j])
            }
        }
        if(p != P-1)
        {
            send computedVal to p+1
        }
        else if(p == P-1  && k == 9)
        {
            print computedVal array
        }
        else if(p == P-1)
        {
            send computedVal to 0
        }
    }
}
```

Memory Consumed: Node 0 creates an array of size `N` to store the computed results and sends this array to the adjacent node. The next node computes values and stores them in the same array. Hence, the total memory consumption is `θ(N)`

Communication per iteration: As this algorithm is CHAIN structured, there are `θ(Communication(N) * P)` communications happening in every iteration. `Communication(N)` implies the communication overhead required to send an array of size `N`.