Jeonghyeon Woo

Cpts 415

HW 3

1. a. Speed-up is expecting less response time when there are more processors, and scale-up is expecting the same response time when there are more processors while there are more tasks.

   I would express:

   Speed-up: $\text{time} = \frac{\text{task}}{\text{processor}} + \text{C}$

   Scale-up: $\text{task} = \text{time} * \text{processor} + \text{C}$

   And, because of the c, which is constant, it is hard to improve the sequence better than linear.

   b. Since we always need to consume at least 0.4t, it will be the base.

   And it will take additional time of 0.6/number_of_processors, so it can be expressed: $f(p) = 0.4t + \frac{0.6}{p}t + \text{c}$, and we ignore c as we find the best time cost.

$$f(1) = 0.4t + \frac{0.6}{1}t = 1t$$

$$f(2) = 0.4t + \frac{0.6}{2}t = 0.7t$$

$$f(4) = 0.4t + \frac{0.6}{4}t = 0.55t$$

$$f(8) = 0.4t + \frac{0.6}{8}t = 0.475t$$

   c. There are three type of parallel system architecture: shared memory, shared disk, and shared nothing.

   Shared memory is putting data in a shared memory. It is easy to program, easy to access data (efficient communication), expensive to build, and difficulty to scaleup.

   Shared memory is putting data in separate memories and accessing each memories' data through io. It has fault tolerance, better scalability, and need of calling io (inefficient communication).

   Shared nothing is sharing nothing. Each processor does its job. It is hard to program (set the task for each processor), cheap to build, and easy to

scaleup.

2. a.

```
cf[]=[]
dic_cf=<(string, string)=primary key, cf_list=[]>
for(i=0; i<data.length; i++)
    for(j=i+1; j<data.length; j++)
        map(data[i],data[j])
for(i=0; i<data.length; i++)
    reduce(data[i])
return dic_cf

map((a,x),(b,y))          //pattern of tuple: data[i]=(a,x) data[j]=(b,y)
    isfriend=0
    for friend in x
        if friend=b then isfriend=1 and break
    if (not isfriend) then stop //if a and b are not friend, then stop mapping
    for a_friend in x and for b_friend in y
        if a_friend=b_friend then cf[(a,x)].append((b,y)) and stop

reduce((a,x))             //pattern of tuple: data[i]=(a,x)
    for(i=0; i<cf[(a,x)].length; i++)
        tuple (b,y) = cf[a,x][i]
        for a_friend in x and for b_friend in y
            if a_friend=b_friend then dic_cf[(a,b)].cf_list.append(temp)
```

b.

```
articles[]=[]
keywords=<keyword=primary key, n_appeared>
for R in data
```

```
        for r in R
            map(R, r)
    for R in data
        reduce(articles[R])
    sort keywords by descending order n_appeared
    return first 10 keyword of keywords.

    map(R, r)
        for keyword in r
            articles[R].append(keyword)

    reduce(R)
        for keyword in R
            keywords[keyword].n_appeard++
```

3. a.

```
cf[]=[]
dic_cf[]=<(string, string)=primary key, cf_list[]>
for(i=0; i<data.length; i++)
    for(j=0; j<data.length; j++)
        map(data[i], data[j])
for(i=0; i<data.length; i++)
    reduce(data[i])
return dic_cf
map(i, j)
    if L_out(i) ∩ L_in(j) then cf[i].append(j)
reduce(i)
    for(j=0; j<cf[i].length; j++)
        for x in L_out(i) ∩ L_out(cf[i][j])
            dic_cf[(i.name, cf[i][j].name)].cf_list.append(x.name)
```

b.

if u is null or v is null or d is null then return false

map(u, v)

if($L_{out}(u) \cap L_{in}(v)$) then emit(u,v)

else return false

reduce((u,v), d)

if(u.DistanceTo(v)<d | u.DistanceTo(v)=d) then return true

//assume properties of adjacency list is imported

else return false

correctness:

For all inputs, if datatype of inputs are Node, Node, and integer, then the program returns bool variable.

Let u = Node1, v = Node2, d=integer

Case 1: u or v or d is null

Program returns bool variable since

$\sim u \cup \sim v \cup \sim d \rightarrow \text{false}$

Case 2: u and v are not reachable to each other

Program returns bool variable since

$$\sim\left(L_{out}(u) \cap L_{in}(v)\right) \rightarrow \ false$$

Case 3: Distance between u and v is greater than d

Program returns bool variable since

$\sim(u.\text{DistanceTo}(v) < d \cup u.\text{DistanceTo}(v) = d) \rightarrow false$

$u.\text{DistanceTo}(v) > d \cap u.\text{DistanceTo}(v) \neq d \rightarrow \text{false}$

$u.\text{DistanceTo}(v) > d \rightarrow \text{false}$ since

$u.\text{DistanceTo}(v) \subset u.\text{DistanceTo}(v) > d$

Case 4: Distance between u and v is less than d or equal to d

Program returns bool variable since

$u.\text{DistanceTo}(v) < d \cup u.\text{DistanceTo}(v) = d \rightarrow \text{true}$

Since all cases of inputs agree that the program returns a bool

variable, (u,v,d){P}(bool) is true.

Complexity: O($|L_{out}(u) \cap L_{in}(v)|$) since worst case is measuring distances of all paths between u and v and the number of the paths is equal to the number of the nodes that both u and v can reach.