

Parallel Programming Principle and Practice

Lecture 10 — parallel algorithm design: examples



parallel algorithm design

examples

Outline

□ Examples

- Relatively simple example
 - ✓ reduction operation
 - ✓ inner product of two vectors
 - ✓ matrix multiplication
- Relatively more complex example
 - ✓ prefix or scan
- *Gaussian elimination with partial pivoting
- *All pairwise computation
- *Sparse matrix computation
- *Graph algorithm

Examples

Relatively simple example

Reduction operation

- ❑ An operator is a **reduction operator** if:
 - It can reduce an array to a single scalar value
 - The final result should be obtainable from the results of the partial tasks that were created
- ❑ Example : Addition
 - ❑ Suppose we have an array [2,3,5,1,7,6,8,4]. The sum of this array can be computed serially by sequentially reducing the array into a single sum using the '+' operator. Starting the summation from the beginning of the array yields

$$\left(\left(\left(\left((2 + 3) + 5 \right) + 1 \right) + 7 \right) + 6 \right) + 8 \right) + 4 = 36$$

How can we compute in parallel way?

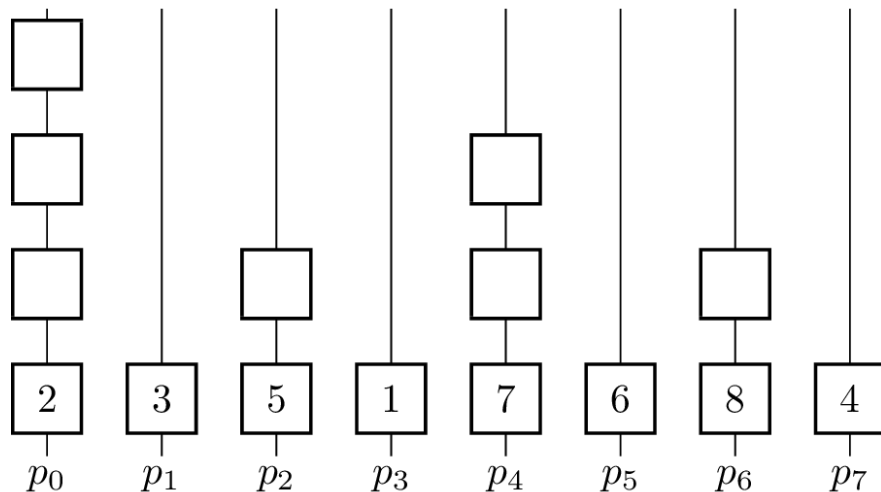
Reduction operation

□ Binomial tree algorithms

- This is an example of **Shared memory algorithm**, for 8 numbers , it just need 3 step time
- Indeed, $O(\log(n))$

□ ‘+’ is both commutative and associative, it is a **reduction operator**

- The commutativity (交换性) of the reduction operator would be important if there were a master core distributing work to several processors, since then the results could arrive back to the master processor in any order. The property of commutativity guarantees that the result will be the same



Reduction operation

- ❑ A reduction operator can help **break down a task into various partial tasks** by calculating partial results which can be used to obtain a final result.
- ❑ It allows **certain serial operations to be performed in parallel** and the number of steps required for those operations to be reduced
- ❑ A reduction operator stores the result of the partial tasks into a private copy of the variable. These private copies are then merged into a shared copy at the end

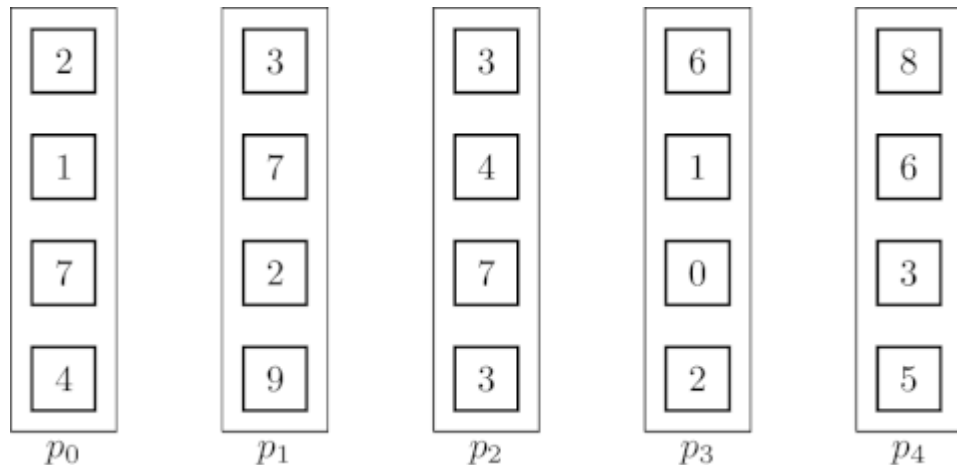
Reduction operation

- the reduction operator is a type of operator that is commonly used in parallel programming to reduce the elements of an array into a single result
 - **The reduction of sets** of elements is an integral part of programming models such as **MapReduce**, where a reduction operator is applied (mapped) to all elements before they are reduced
 - Other parallel algorithms use reduction operators as primary operations to solve more complex problems
 - Many reduction operators can be used for broadcasting to distribute data to all processors

Reduction operation

□ Pipeline-algorithm

- In the distributed memory model, memory is not shared between processing units and data has to be exchanged explicitly between processing units.
- Therefore, data has to be exchanged explicitly between units, as can be seen in the algorithm graph



We need to compute $p_0+p_1+p_2+p_3+p_4$. Each p is in an independent memory unit. At the same time, **only one exchange can happen between two memory units**

inner product of two vectors

- ❑ In mathematics, the **dot product** or **scalar product** is an algebraic operation that takes two equal-length sequences of numbers (usually coordinate vectors), and returns a single number
- ❑ In Euclidean geometry, the dot product of the Cartesian coordinates of two vectors is widely used
- ❑ It is often called the **inner product** (or rarely **projection product**) of Euclidean space

Matrix Representation of Dot Product

$$\vec{A}^T = \begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \quad \vec{B} = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

$$\begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = A_1B_1 + A_2B_2 + A_3B_3 = \vec{A} \cdot \vec{B}$$

inner product of two vectors

- ❑ We can compute inner product of two vectors in a parallel way
- ❑ As the example as follows, we can compute it with 3 threads

$$\begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \underbrace{A_1 B_1}_{\text{thread1}} + \underbrace{A_2 B_2}_{\text{thread2}} + \underbrace{A_3 B_3}_{\text{thread3}} = \vec{A} \cdot \vec{B}$$

思考:

- lock的作用域范围?
- 如果vector的长度n很大, 如何避免线性时间的结果归并?

```
#include <iostream>
#include <thread>
#include <Windows.h>
#include <mutex>
using namespace std;
mutex mut;
mutex mut;
int sum = 0; //共享变量
void thread_mul(int i)
{
    mut.lock(); //将互斥锁进行lock
    int mulRes = A[i]B[i];
    sum += mulRes;
    mut.unlock(); //unlock 解开互斥锁
}
int main()
{
    thread task1(thread_mul,0);
    thread task2(thread_mul,1);
    thread task3(thread_mul,2);
    task1.join();
    task2.join();
    task3.join()
    cout << "main thread!" << endl;
}
```

matrix multiplication : Data parallelism

- ❑ matrix multiplication is a typical example of data parallelism
- ❑ Data parallelism is **parallelization across multiple processors** in parallel computing environments
 - It focuses on distributing the data across different nodes, which operate on the data in parallel
 - It can be applied on regular data structures like arrays and matrices by working on each element in parallel

matrix multiplication

- In mathematics, particularly in linear algebra, **matrix multiplication** is a binary operation that produces a matrix from two matrices
- If \mathbf{A} is an $m \times n$ matrix and \mathbf{B} is an $n \times p$ matrix, the matrix product $\mathbf{C} = \mathbf{AB}$ is defined to be the $m \times p$ matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj},$$

matrix multiplication : sequential

- Below is the sequential code for multiplication calculates the dot product of two matrices A, B and stores the result into the output matrix C.

```
// Matrix multiplication
for (i = 0; i < row_length_A; i++)
{
    for (k = 0; k < column_length_B; k++)
    {
        sum = 0;
        for (j = 0; j < column_length_A; j++)
        {
            sum += A[i][j] * B[j][k];
        }
        C[i][k] = sum;
    }
}
```

matrix multiplication : parallel with $m*k$ processors

- ❑ divide matrix A and B into blocks along rows and columns respectively
- ❑ calculate every element in matrix C individually thereby making the task parallel
- ❑ $A[m \times n] \text{ dot } B[n \times k]$ can be finished in $O(n)$ instead of $O(m * n * k)$ when executed in parallel using $m*k$ processors

```
// Matrix multiplication in parallel
```

```
#pragma omp parallel for schedule(dynamic,1) collapse(2)
```

```
for (i = 0; i < row_length_A; i++){
```

```
    for (k = 0; k < column_length_B; k++){
```

```
        sum = 0;
```

```
        for (j = 0; j < column_length_A; j++){
```

```
            sum += A[i][j] * B[j][k];
```

```
        }
```

```
        C[i][k] = sum;
```

```
    }
```

```
}
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 3 & 2 \end{pmatrix}$$

3 x 3

$$\begin{pmatrix} 10 & 11 \\ 7 & 5 \\ 2 & 4 \end{pmatrix}$$

3 x 2

$$= \begin{pmatrix} 1*10+2*7+3*2 & 1*11+2*5+3*4 \\ 4*10+5*7+6*2 & 4*11+5*5+6*4 \\ 1*10+3*7+2*2 & 1*11+3*5+2*4 \end{pmatrix}$$

3 x 2

Sum需要临界保护么?

matrix multiplication : OpenCL

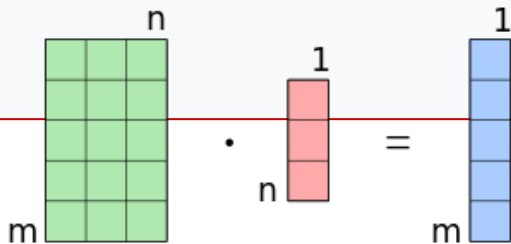
□ matrix-vector multiplication algorithm in OpenCL C

```
// Multiplies A*x, leaving the result in y.
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
__kernel void matvec(__global const float *A, __global const float *x,
                    uint ncols, __global float *y)
{
    size_t i = get_global_id(0);           // Global id, used as the row index
    __global float const *a = &A[i*ncols]; // Pointer to the i'th row
    float sum = 0.f;                        // Accumulator for dot product
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

思考:

- 此时处理器个数 $p=?$

$$y_i = a_{i,:} \cdot x = \sum_j a_{i,j} x_j.$$



Examples

Relatively more complex example

prefix or scan : prefix sum

□ Prefix sum is a typical example of prefix or scan:

for array: $a[4] = \{1, 2, 3, 4\}$;

prefix sum:

$$S[0] = a[0] = 1;$$

$$S[1] = a[0] + a[1] = 1 + 2 = 3;$$

$$S[2] = a[0] + a[1] + a[2] = 1 + 2 + 3 = 6;$$

$$S[3] = a[0] + a[1] + a[2] + a[3] = 1 + 2 + 3 + 4 = 10;$$

简单来说:

- Prefix是啥?

Prefix sum : static load balancing algorithm

□ static load balancing algorithm

- A load balancing algorithm is "static"
 - ✓ when it does not take into account the state of the system for the distribution of tasks,
 - the system state includes measures such as the **load level of certain processors**
 - ✓ Instead, assumptions about the overall system are made beforehand,
 - such as the arrival times and resource requirements of incoming tasks
- Therefore, static load balancing **aims to**
 - ✓ associate **a known set of tasks** with **the available processors** in order to minimize a certain performance function

Prefix sum : static load balancing algorithm

□ static load balancing algorithm

- Prefix sum is a typical example of **static load balancing algorithm**
 - ✓ By dividing the tasks to give the same amount of computation to each processor, all that remains to be done is to group the results together
 - ✓ Using a prefix sum algorithm, **this division can be calculated in logarithmic time** with respect to the number of processors

- 一定可以做到么？
- 最差的复杂度可能是多少？

Prefix sum : serial & parallel way

□ For serial:

$$S[i+1]=S[i]+a[i+1]$$

□ For parallel:

For 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Step1: divide into four arrays and give the same amount of computation to each processor and then calculate their prefix sum

(1 2 3 4) (5 6 7 8) (9 10 11 12) (13 14 15 16)

(1 3 6 10) (5 11 18 26) (9 19 30 42) (13 27 42 58)

Step2: Choose the last number of the result of Step2 and calculate the sum:

(1 3 6 10) (5 11 18 36) (9 19 30 78) (13 27 42 136)

~~36=10+26~~

~~78=10+26+42~~

Examples

*Gaussian elimination with partial pivoting
(部分置换高斯消去法)

*Gaussian elimination with partial pivoting (部分置换高斯消去法)

□ One of the most popular techniques for solving simultaneous linear equations of the form

➤ Consists of 2 steps

- Forward Elimination of Unknowns
 - ✓ transform the coefficient matrix into an Upper Triangular Matrix
- Back Substitution
 - ✓ solve each of the equations using the upper triangular matrix

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

*Gaussian elimination with partial pivoting

□ we have order of magnitude differences between coefficients in the different rows

$$\begin{bmatrix} 0.02 & 0.01 & 0 & 0 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 100 & 200 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 1 \\ 4 \\ 800 \end{bmatrix} \Rightarrow \left[\begin{array}{cccc|c} 0.02 & 0.01 & 0 & 0 & 0.02 \\ 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right]$$

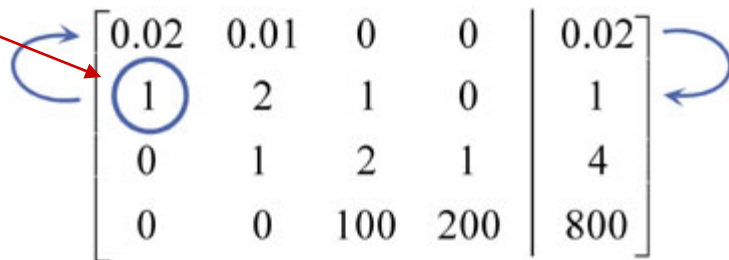
✓ 1. **Gaussian elimination**(高斯消元)

✓ 2. **Back substitution**(回代)

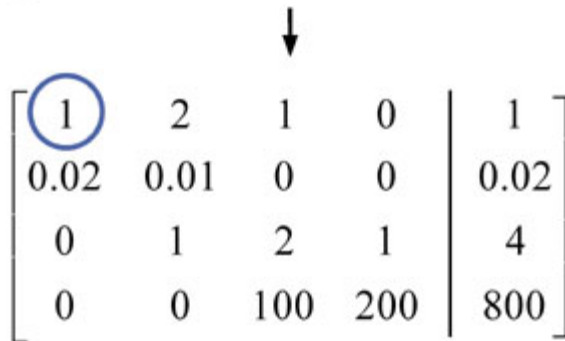
Number of step of Gaussian elimination is $(n-1) = 4-1 = 3$

*Gaussian elimination with partial pivoting

- Step 0a: Find the entry in the left column with the largest absolute value. This entry is called **the pivot**
- Step 0b: Perform **row interchange** (if necessary), so that the pivot is in the first row


$$\left[\begin{array}{cccc|c} 0.02 & 0.01 & 0 & 0 & 0.02 \\ 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right]$$

↓


$$\left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0.02 & 0.01 & 0 & 0 & 0.02 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right]$$

pivot is in the first row

*Gaussian elimination with partial pivoting

□ Step 1: Gaussian Elimination

$$r2 = r1 * (-0.02) + r2$$

1	2	1	0	1
0.02	0.01	0	0	0.02
0	1	2	1	4
0	0	100	200	800



1	2	1	0	1
0	-0.03	-0.02	0	0
0	1	2	1	4
0	0	100	200	800

*Gaussian elimination with partial pivoting

- Step 2: Find new pivot
- Step 3: Switch rows (if necessary)

$$\begin{array}{c} \curvearrowright \left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & -0.03 & -0.02 & 0 & 0 \\ 0 & \textcircled{1} & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right] \curvearrowright \\ \downarrow \\ \left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & \textcircled{1} & 2 & 1 & 4 \\ 0 & -0.03 & -0.02 & 0 & 0 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right] \end{array}$$

*Gaussian elimination with partial pivoting

□ Step 4: Gaussian Elimination

$$\left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & -0.03 & -0.02 & 0 & 0 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right]$$

$$r3 = r2 * (0.03) + r3$$

$$\downarrow$$
$$\left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 0.04 & 0.03 & 0.12 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right]$$

*Gaussian elimination with partial pivoting

- Step 5: Find new pivot
- Step 6: Switch rows (if necessary)

$$\begin{array}{c} \curvearrowright \left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 0.04 & 0.03 & 0.12 \\ 0 & 0 & 100 & 200 & 800 \end{array} \right] \curvearrowleft \end{array}$$

↓

$$\begin{bmatrix} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \\ 0 & 0 & 0.04 & 0.03 & 0.12 \end{bmatrix}$$

*Gaussian elimination with partial pivoting

□ Step 7: Gaussian Elimination

$$\left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \\ 0 & 0 & 0.04 & 0.03 & 0.12 \end{array} \right]$$

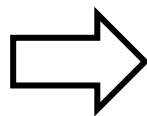
$$r_4 = r_3 * (-0.04/100) + r_4$$

$$\left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \\ 0 & 0 & 0 & -0.05 & -0.2 \end{array} \right]$$

*Gaussian elimination with partial pivoting

□ Step 8: **Back Substitute**

$$\left[\begin{array}{cccc|c} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \\ 0 & 0 & 0 & -0.05 & -0.2 \end{array} \right]$$



$$\begin{aligned} -0.05 * X_4 &= -0.2; X_4 = 4 \\ 100 * X_3 + 200 * X_4 &= 800; X_3 = 0 \\ X_2 + 2 * X_3 + X_4 &= 4; X_2 = 0 \\ X_1 + 2 * X_2 + X_3 &= 1; X_1 = 1 \end{aligned}$$

*Gaussian elimination with partial pivoting

- 它是Prefix algorithm么?
- How to parallel?

*Gaussian elimination with partial pivoting

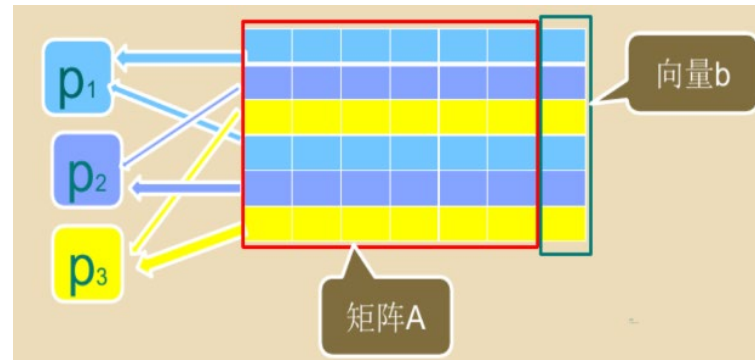
□ How to parallel?

➤ 找最大元

- 每次消去计算前，各处理器并行求其局部存储器中右下角子阵的最大元
- 记录其所在的行号、列号及所在的处理器的编号。并把信息广播给所有的处理器

➤ 交换行列

- 接受消息，求出右下角子阵的最大元 maxvalue 及其所在的行号、列号及所在处理器的编号
- ✓ 列交换
 - 当 maxvalue 的列号不是原主元素 a 的列号时，则交换两列数据
- ✓ 行交换
 - maxvalue 的所在的处理器的编号是原主元素 a 的处理器的编号时，则要在处理器内部交换
 - maxvalue 的所在的处理器的编号不是原主元素 a 的处理器的编号时，则要在处理器间交换



*Gaussian elimination with partial pivoting

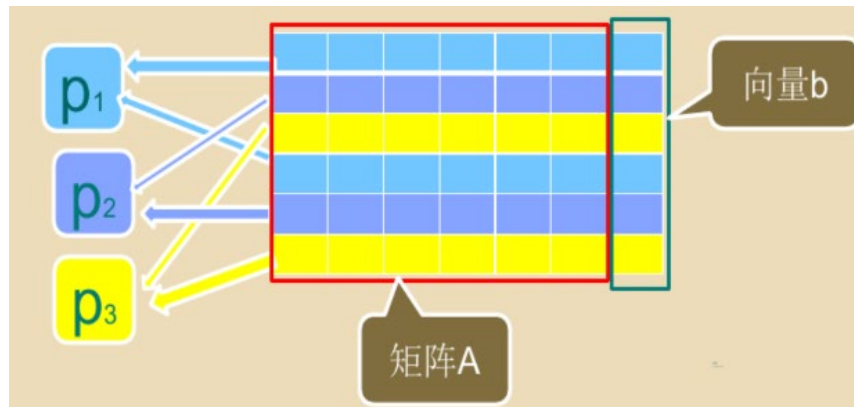
□ How to parallel?

➤ 消去计算

- ✓ 对主元素作归一化操作，将主行广播给所有处理器，各处理器利用接收到的主元素对其部分行向量做行变换

➤ 回代过程

- ✓ 一旦被计算出来，立即广播给所有的处理器，用于与对应项相乘并做求和计算



*Gaussian elimination with partial pivoting

□ Practice

➤ 已知线性方程组

$$2x + y - z = 8$$

$$-3x - y + 2z = -11$$

$$-2x + y + 2z = -3$$

➤ 试求解 x, y, z 未知数的值

*Gaussian elimination with partial pivoting

并行计算过程:

1. 第1行分给1号进程, 第2行分给2号进程,, 第P行分给P号进程

$$\begin{array}{l} \mathbf{P1} \leftarrow \\ \mathbf{P2} \leftarrow \\ \mathbf{P3} \leftarrow \end{array} \left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array} \right)$$

(A|b)增广矩阵

2. 求得第一行的主元(一般选择第一个元素), 并将第一行所有元素及所在位置发送给每个进程, 每个进程按照该行元素以及主元位置将各自进程中对应位置消为0, 直到第二行, 第三行完成操作

① P1进程选择第一行的主元2, 并记录其位置1(列), 传递给P2,P3进程

$$\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array} \right)$$

*Gaussian elimination with partial pivoting

② P2,P3进程按照P1进程传递的第一行中所有元素以及第一行主元位置，将各自进程中对应位置消为0

P1 $\leftarrow \begin{pmatrix} 2 & 1 & -1 & | & 8 \end{pmatrix}$

P2 $\leftarrow \begin{pmatrix} 0 & 1/2 & 1/2 & | & 1 \end{pmatrix}$

P3 $\leftarrow \begin{pmatrix} 0 & 2 & 1 & | & 5 \end{pmatrix}$

③ P2进程选择第二行的主元1/2，并记录其位置2(列),传递给P1,P3进程

$$\begin{pmatrix} 2 & 1 & -1 & | & 8 \\ 0 & \boxed{1/2} & 1/2 & | & 1 \\ 0 & 0 & -1 & | & 1 \end{pmatrix}$$

④ P3进程按照P1进程传递的第二行中所有元素以及第二行主元位置，将各自进程中对应位置消为0

P1 $\leftarrow \begin{pmatrix} 2 & 1 & -1 & | & 8 \end{pmatrix}$

P2 $\leftarrow \begin{pmatrix} 0 & 1/2 & 1/2 & | & 1 \end{pmatrix}$

P3 $\leftarrow \begin{pmatrix} 0 & 0 & -1 & | & 1 \end{pmatrix}$

*Gaussian elimination with partial pivoting

⑤ P1,P2,P3处理器按照各自主元情况，将主元进行归一化操作，同时每行另外元素的大小会随之变化

$$\begin{array}{lcl}
 \text{P1} & \leftarrow & \left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \end{array} \right) \\
 \text{P2} & \leftarrow & \left(\begin{array}{ccc|c} 0 & 1/2 & 1/2 & 1 \end{array} \right) \\
 \text{P3} & \leftarrow & \left(\begin{array}{ccc|c} 0 & 0 & -1 & 1 \end{array} \right)
 \end{array}
 \longrightarrow
 \left(\begin{array}{ccc|c} 1 & 1/2 & -1/2 & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array} \right)$$

3.回代:P1,P2,P3进程进行各自行求未知值过程，一旦被计算出来，立即广播给所有的处理器，用于与对应项相乘并做求和计算

$$\begin{cases}
 z = -1 & \leftarrow \text{P3处理器求出} Z=-1, \text{立即广播给P1,P2处理器} \\
 y = 2 - z = 3 & \leftarrow \text{P2处理器收到P3广播的} Z=-1 \text{信息, 求出} Y=3, \text{立即广播给P1,P3处理器} \\
 x = 4 + \frac{1}{2}z - \frac{1}{2}y = 2 & \leftarrow \text{P1处理器收到P3广播的} Z=-1 \text{信息,P2广播的} Y=3 \text{信息, 求出} X=2, \text{运算结束}
 \end{cases}$$

Examples

- *All pairwise computation (全成对计算)

*All pairwise computation (全成对计算)

- Be defined as performing computation (e.g., correlations) between every pair of the elements in a given dataset
- commonly used in various kinds of applications in science and engineering
- Assume that the number of sequences is n , each being of length m
 - be a total number of $n(n-1)/2$ sequence pairs
 - the computational complexity is thus $O(n^2 f(m))$

*All pairwise computation

□ Example: Pairwise distance

➤ Euclidean Distance

- ✓ two-dimensional:

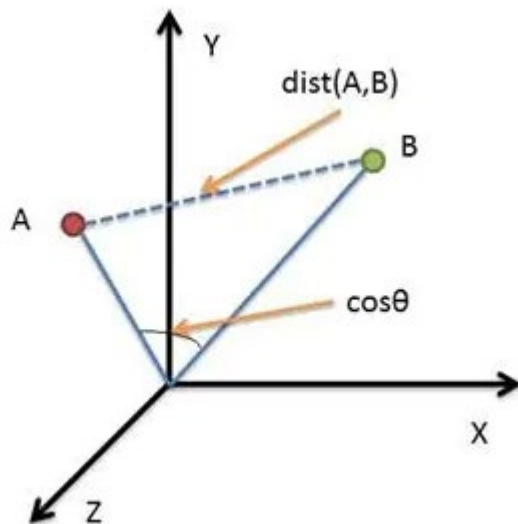
$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- ✓ three-dimensional:

$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

- ✓ n-dimensional:

$$d_{12} = \sqrt{\sum_{k=1}^n (x_{1k} - x_{2k})^2}$$



*All pairwise computation (全成对计算)

- 可以用reduce algorithm求解么?
- How to parallel?

*All pairwise computation (全成对计算)

□ How to parallel?

➤ 共享内存机器

- ✓ 将一组作为一个任务分配给一个计算节点

➤ 使用主/工作者并行编程范例

- I. **序列首先被划分成块**。单个主节点向工作节点发送任务，每次一个
- II. 当接收到一个任务时，worker从I/O磁盘加载两个相关联的序列块
- III. 进行计算，将结果发送回主服务器
- IV. 然后等待主服务器的另一个任务

Examples

*Sparse matrix computation

*Sparse matrix computation

□ Sparse matrix

➤ In numerical analysis and scientific computing, a **sparse matrix** or sparse array is a matrix in which most of the elements are zero

- If most of the elements are non-zero, the matrix is considered **dense**
- The density of the matrix : The number of Nonzero-valued elements divided by the total number of elements
- The sparsity of the matrix : The number of zero-valued elements divided by the total number of elements

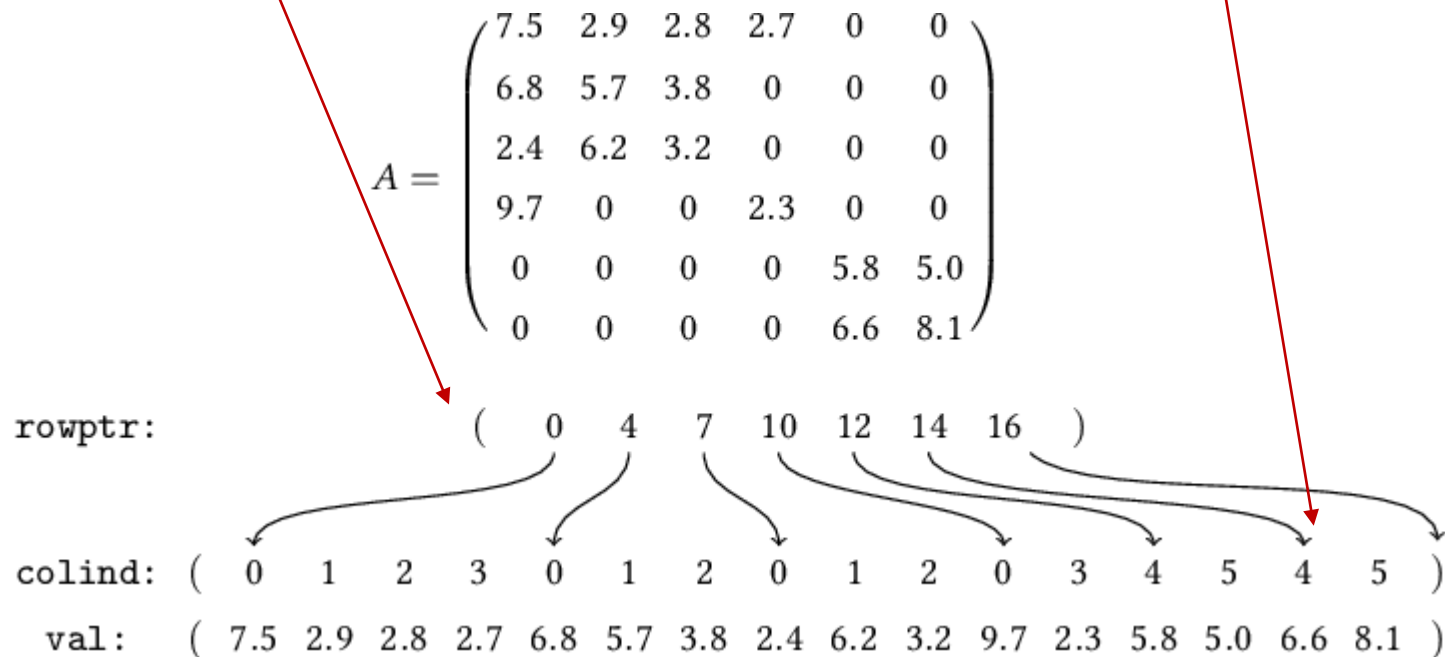
Example of sparse matrix

$$\begin{pmatrix} 11 & 22 & 0 & 0 & 0 & 0 & 0 \\ 0 & 33 & 44 & 0 & 0 & 0 & 0 \\ 0 & 0 & 55 & 66 & 77 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 88 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 99 \end{pmatrix}$$

The above sparse matrix contains only 9 non-zero elements, with 26 zero elements. Its sparsity is 74%, and its density is 26%.

Sparse matrix computation with Compressed Sparse Row Storage

- **Compressed Sparse Row (CSR) format** uses a row pointer structure to index the start of each row within the array of nonzero elements, and a column index structure to store the column of each nonzero element



*Sparse matrix computation

a) Matrix M

3	4	0	0
0	5	9	0
2	0	3	1
0	4	0	6

b) values

3	4	5	9	2	3	1	4	6
---	---	---	---	---	---	---	---	---

columnIndex

0	1	1	2	0	2	3	1	3
---	---	---	---	---	---	---	---	---

rowPtr

0	2	4	7	9
---	---	---	---	---

```
#include "spmv.h"
```

```
void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
```

```
DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE]){
```

```
L1: for (int i = 0; i < NUM_ROWS; i++) {
```

```
    DTYPE y0 = 0;
```

```
    L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
```

```
        #pragma HLS unroll factor=8
```

```
        #pragma HLS pipeline
```

```
        y0 += values[k] * x[columnIndex[k]];
```

```
    }
```

```
    y[i] = y0;
```

```
}
```

```
}
```

❑ 稀疏矩阵向量乘 (SpMV) $y=M*x$ 的计算

- 第一个for循环通过迭代访问每一行
- 第二个for循环访问每一列
- 实现矩阵M中非0元素和向量中对应的元素相乘并保存值在向量y中

Sparse matrix computation with Compressed Sparse Row Storage

➤ How to parallel ?

Sparse matrix computation : parallel example

- Here is an example of multiplication of sparse matrix and a vector.

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix} \quad x = [1 \ 1 \ 1 \ 1 \ 1]^T \quad b = Ax$$

Compressed Sparse Row (CSR) format of A is as follows:

$$A = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)^T$$

$$JA = (1, 4, 1, 2, 4, 1, 3, 4, 5, 3, 4, 5)^T$$

$$IA = (1, 3, 6, 10, 12, 13)^T$$

$$b = (3, 12, 30, 21, 12)^T$$

Sparse matrix computation : parallel example

- Here is an example of serial program with C++. $y = Ax$

```
for (int i = 0; i < n; i++) {  
    k1 = IA.at(i);  
    k2 = IA.at(i + 1);  
    sum = 0.0;  
    for (int j = k1; j < k2; j++)  
    {  
        sum = sum + A.at(j - 1) * x.at(IA.at(j - 1) - 1);  
    }  
    y[i] = sum;  
}
```

Sparse matrix computation : parallel example

➤ We can compute it **in parallel** way with multiple threads

- ✓ Divide multiple numbers of a sparse matrix into multiple threads for calculation

Step1: package the serial code for every thread.

```
/**  
将稀疏矩阵分为多个部分并行执行  
**/  
//将串行执行程序打包封装  
auto Task = [this, &y, &x](int idx1, int idx2) {  
    int k1, k2;  
    double sum;  
    for (int i = idx1; i < idx2; i++) {  
        //第 idx1 行的数存储在A中的k1到k2中  
        k1 = IA.at(i);  
        k2 = IA.at(i + 1);  
        sum = 0.0;  
        for (int j = k1; j < k2; j++)  
        {  
            sum = sum + A.at(j - 1) * x.at(JA.at(j - 1) - 1);  
        }  
        //y存储最终结果  
        y[i] = sum;  
    }  
};
```

Sparse matrix computation : parallel example

- Step2: Assuming thread pool has *pnumbers* thread, We divide the numbers of sparse matrix into *pnumbers* parts and put it in threads.

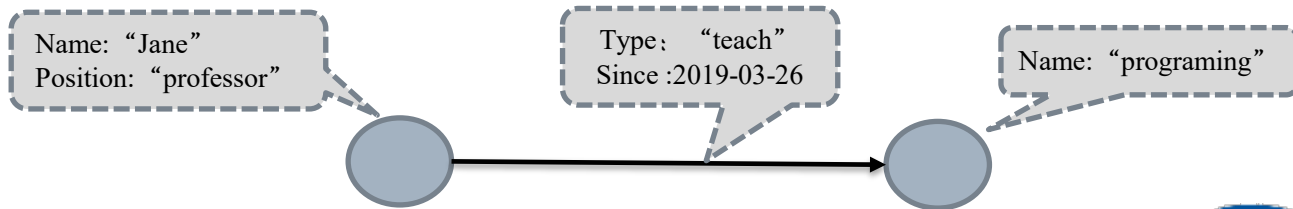
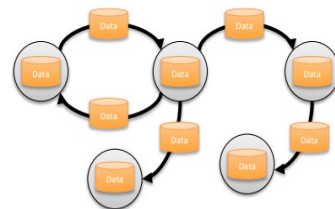
```
//获取线程池中线程数目
int pnumbers = thread_pool->ReturnThreadNumbers();
//fu 用于保存线程
vector<future<void>> fu;
//n 为稀疏矩阵中的行数，将这些计算平均分给 pnumbers 个线程，每个线程负责计算 interp 行
int interp = static_cast<int>(floor(n / static_cast<double>(pnumbers)));
//将这些计算的参数传输给线程
for (int i = 0; i < pnumbers - 1; i++)
{
    fu.emplace_back(thread_pool->submit(Task, i * interp, (i + 1) * interp));
}
fu.emplace_back(thread_pool->submit(Task, interp * (pnumbers - 1), n));
//启动每一个线程进行计算
for (int i = 0; i < pnumbers; i++)
{
    fu.at(i).get();
}
```

Examples

*Graph algorithm

What is Graph?

- A graph is a collection of **vertices** connected by **edges** ($G = (V, E)$)
- Edge may be **directed** (from A to B) or **undirected** (between A and B)
- Vertices and edges may have **properties**



Traditional Processing vs Graph Processing



■ Compute intensive

Computation complexity is high, easy to cover
memory latency

■ Sequential memory access

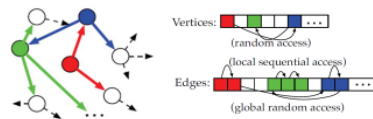
Data is stored in memory sequentially

■ High data parallelism

No complex dependency between different data,
convenient for parallel processing

■ Good locality

With good spatial and temporal locality



■ Low computation/access ratio

Little computation in each node, hard to cover latency

■ Large random access

There is a large number of random access requests across the region

■ Complex data dependency

Difficult to explore parallelism, a lot of data conflicts

■ Unstructured distribution

Uneven distribution of vertex degree, serious load balancing problems and communication overhead

Graph Algorithm

- **Traversal-centric algorithms**

- Require a specific way to traverse the graph from a particular vertex
- Have a large number of random access

- **Computation-centric algorithms**

- A large number of operations in each iteration
- All nodes participate in each iteration

BFS	Dijkstra	Prim	SSSP	Betweenness Centrality	Radii
DFS	MST	Kruskal	Bellman Ford	Floyd Warshall	SPFA

Traversal-centric algorithms

A*搜索	PageRank	Connected Component	Ford Fulkerson
Union Find	Graph Color	MIS	Triangle Count

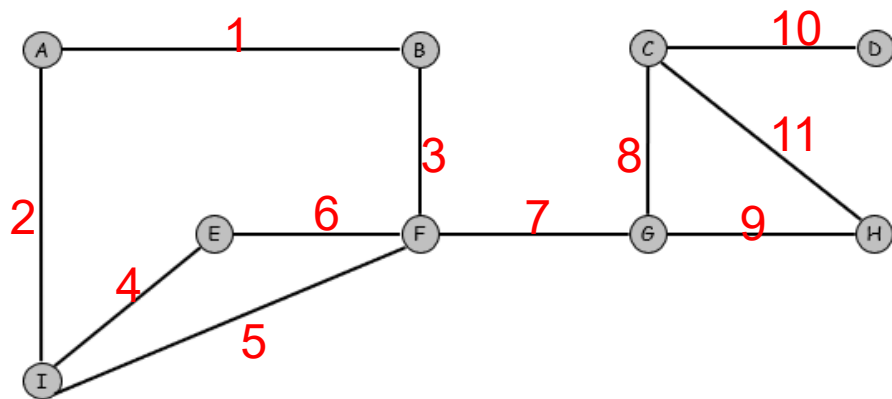
Computation-centric algorithms



*Graph algorithm Example: Breadth First Search(BFS)

□ Algorithm steps:

- ① put the root node in the queue.
- ② Take the first node from the queue and verify that it is the target.
- ③ If the target is found, the search is ended and the result is returned.
- ④ Otherwise, all its direct child nodes that have not been verified will be added to the queue
- ⑤ If the queue is empty, it means that the whole graph has been checked - that is, there is no target to search in the graph. End the search and return "target not found"
- ⑥ Repeat step 2

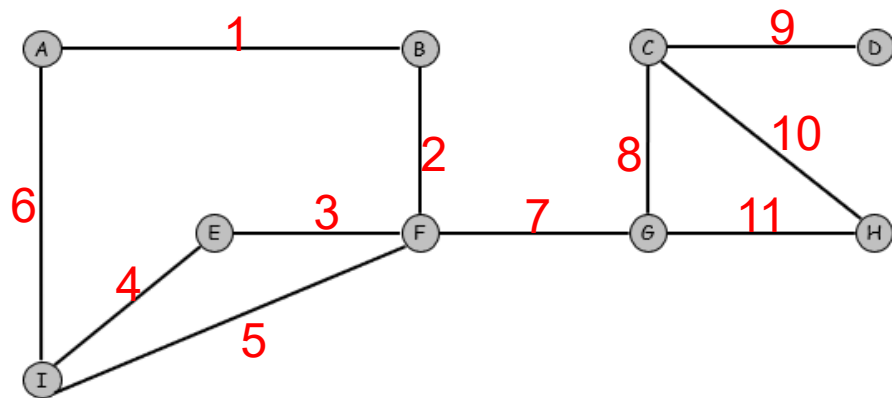


如果从A开始想找到H节点
依次处理“节点”顺序为：A B I F E G C D H
依次处理“边”如图中序号所示

*Graph algorithm Example: Depth First Search(DFS)

□ Algorithm steps:

- ① First, find the initial node A
- ② Based on this, the depth first traversal of the graph is carried out from the adjacent points of a that are not accessed
- ③ If a node is not accessed, it will be traced back to the node and continue the depth first traversal
- ④ Until all nodes figured out with vertex a path have been accessed once



如果想从A开始遍历所有的边
依次处理“节点”顺序为：A B I F E G C D H
依次处理“边”如图中序号所示

*Graph algorithm Example: Parallel example for BFS

□ Serial BFS

- In the conventional sequential BFS algorithm, **two data structures** are created to store the **frontier** and **the next frontier**
 - ✓ The frontier contains the vertexes that have same distance (also called "level") from the source vertex, these vertexes need to be explored in BFS
 - ✓ Every neighbor of these vertexes will be checked, some of these neighbors which are not explored yet will be discovered and put into the next frontier

*Graph algorithm Example: Parallel example for BFS

□ Serial BFS

- The following pseudo-code outlines the idea of it, in which the data structures for the **frontier** and are **next frontier** called **FS** and **NS** respectively

```
1  define bfs_sequential(graph(V,E), source s):
2      for all v in V do
3          d[v] = -1;
4      d[s] = 0; level = 1; FS = {}; NS = {};
5      push(s, FS);
6      while FS != empty do
7          for u in FS do
8              for each neighbour v of u do
9                  if d[v] = -1 then
10                     push(v, NS);
11                     d[v] = level;
12             FS = NS, NS = {}, level = level + 1;
```

- At the beginning of BFS algorithm,
 - ✓ a given **source vertex s** is the only vertex in the frontier
 - ✓ All direct neighbors of s are visited in the first step, which form the **next frontier**
- After each layer-traversal, the "**next frontier**" is switched to the frontier and new vertexes will be stored in the new next frontier

*Graph algorithm Example: Parallel example for BFS

□ Parallel BFS in **distributed memory model**

- In the distributed memory model, each processing entity has its own memory
 - ✓ Because of this, processing entities must send and receive messages to each other to share its local data or get access to remote data
- The neighbor vertex from one processor may be stored in another processor
 - ✓ As a result, each processor is responsible to tell those processors about traversal status through sending them messages
- Moreover, each processor should also deal with the messages from all other processors to construct its local next vertex frontier
- Obviously, one all-to-all communication (which means each entity has different messages for all others) is necessary in each step when exchanging the current frontier and the next vertex frontier

Parallel example for BFS

□ Parallel BFS

- ✓ Combined with multi-threading, the following pseudo code shows the parallel BFS with distributed memory

```
/**
用于分布式存储的多处理器并行BFS
每个具有分布式内存的处理器应该负责大致相同数量的顶点及其出边。
**/
//每个线程从点图中的某个点s开始进行BFS
define parallel_distributed_memory_BFS( graph(V,E), source s):
    //初始化
    for all v in V do
        d[v] = -1;
    //s点的层数为0
    d[s] = 0; level = 0; FS = {}; NS = {};
    push(s, FS);
    //开始BFS
    while FS != empty do:
        //找出FS中所有点的所有邻居节点
        NS = {neighbors of vertexes in FS without level, both local and not local vertexes}
        //与所有的其他p个处理器进行通信
        for 0 <= j < p do:
            N_j = NS
            //将自己的NS告知给其他处理器
            send N_j to processor j
            //接收来自其它处理器的邻边信息
            receive N_j_rcv from processor j
        //将自己的NS与其它处理器的NS结合，总结所有包含在本线程所负责的点集中的点
        NS_rcv = Union(N_j_rcv)
        //设置这些点的层数
        FS = {}
        for v in NS_rcv and d[v] == -1 do
            d[v] = level + 1
            FS.add(v)
        NS = {}
        level++
```

Conclusion

□ Examples

- Relatively simple example
 - ✓ reduction operation
 - ✓ inner product of two vectors
 - ✓ matrix multiplication
- Relatively more complex example
 - ✓ prefix or scan
- *Gaussian elimination with partial pivoting
- *All pairwise computation
- *Sparse matrix computation
- *Graph algorithm