A 1st introduction to
High Performance Computing
Concepts & Techniques

10 Other Computation examples

孔令波
mlinking@126.com
+86 15010255486

大纲

前言

- 为什么需要“大规模计算” [HPC, DL, Business platform system, Cloud已经合流]
  - 导入 – 科学计算(天气预报)，DL, 互联网平台(Google, Amazon, Alibaba, MeiTuan, …)

基础篇

- 并发程序的样子 – Divide & Conquer, Model & Challenges, PCAM, Data/Task, …
  - 天气预报的计算
- 运行环境
  - 硬件 – 自己梳理的3个方案 – Shared/Non-shared Memory, Hybrid
  - 系统软件 – 协议栈, Modern OS, Distributed Job Scheduler, GTM等

算法级篇

- OpenMP, MPI, CUDA (DL的实现), Big Data 中的MR/Spark等 (只涉及在Big Data SDK之上的编程；大数据本身的介绍放到后一部分)
  - Python: Numba (OpenMP), MPI4PY (MPI), PyCUDA, PySpark

系统级篇 – 互联网平台的实现

- “秒杀”的技术架构; 计算广告; 系统架构 (HTAP等)
  - Flink, ClickHouse, MaxCompute, ELK …

10 Other Computation examples

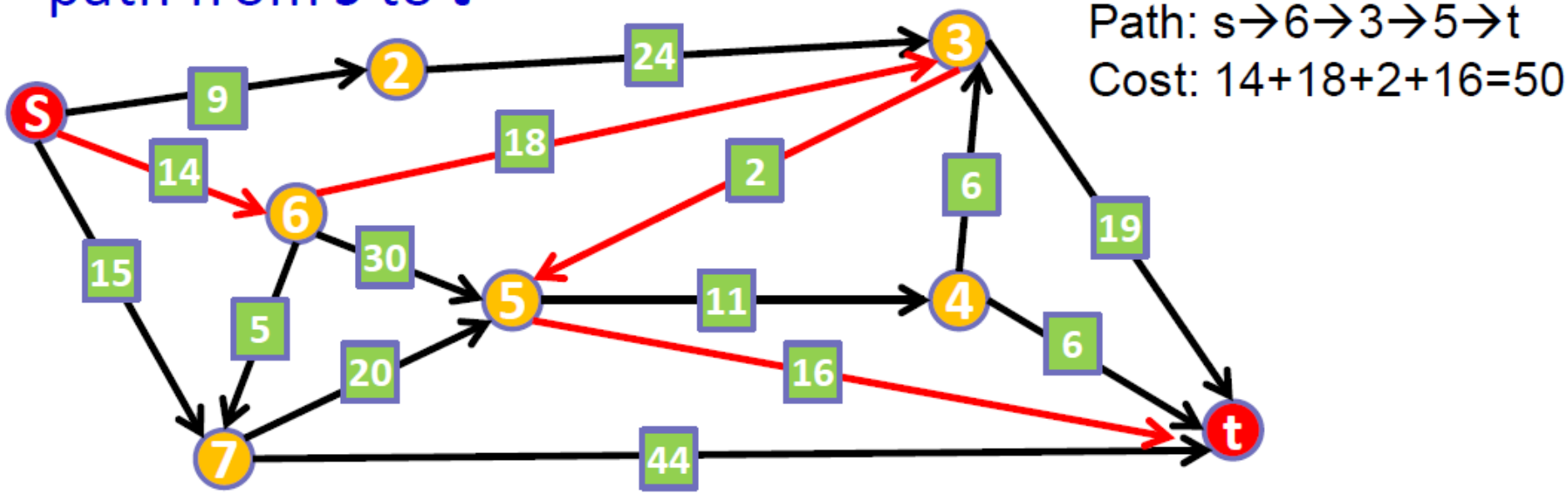
- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

Shortest Path in a Weighted Diagraph

- Given a weighted graph, find the shortest directed path from ***s*** to ***t***



- Note: weight are arbitrary numbers
  - Not necessary distance
  - Need not satisfy the triangle inequality

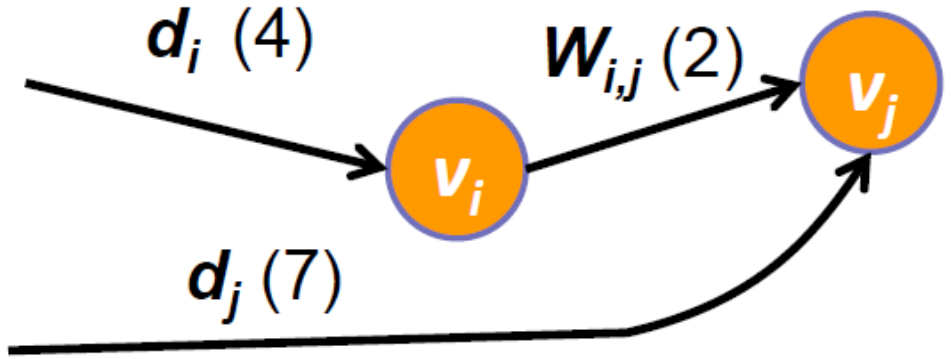
- Shimbel (1955): Information networks
- Ford (1956): RAND, economics of transportation
- Dantzig (1958): Simplex method for linear programming
- Bellman (1958): Dynamic programming
- Moore (1959): Routing long-distance phone calls from Bell Lab
- Dijkstra (1959): Simpler and faster version of Ford's algorithm

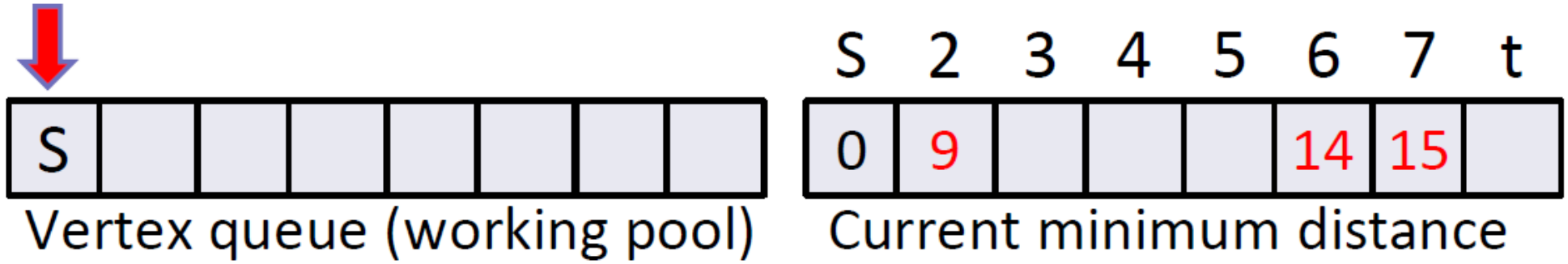
- We choose Moore because it is more amenable to parallel implementation

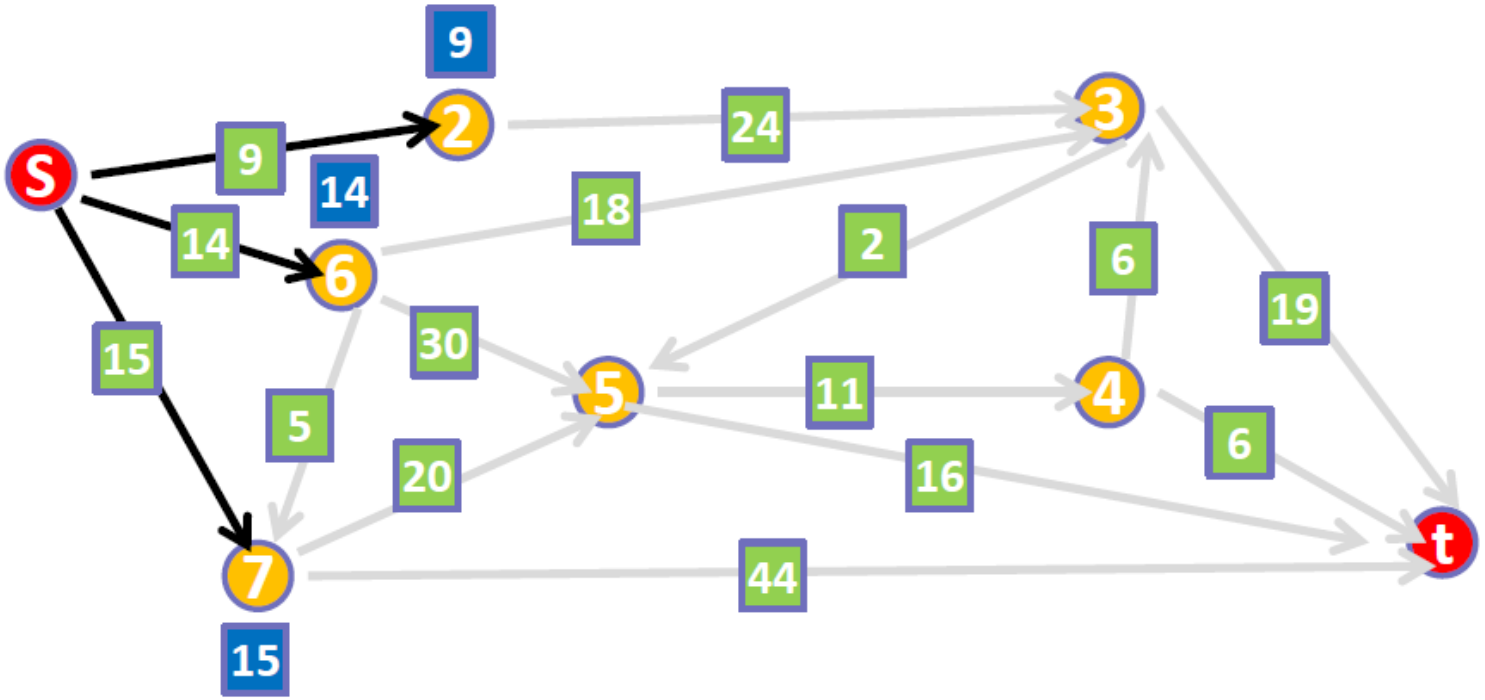
Moore's Shortest Path Algorithm

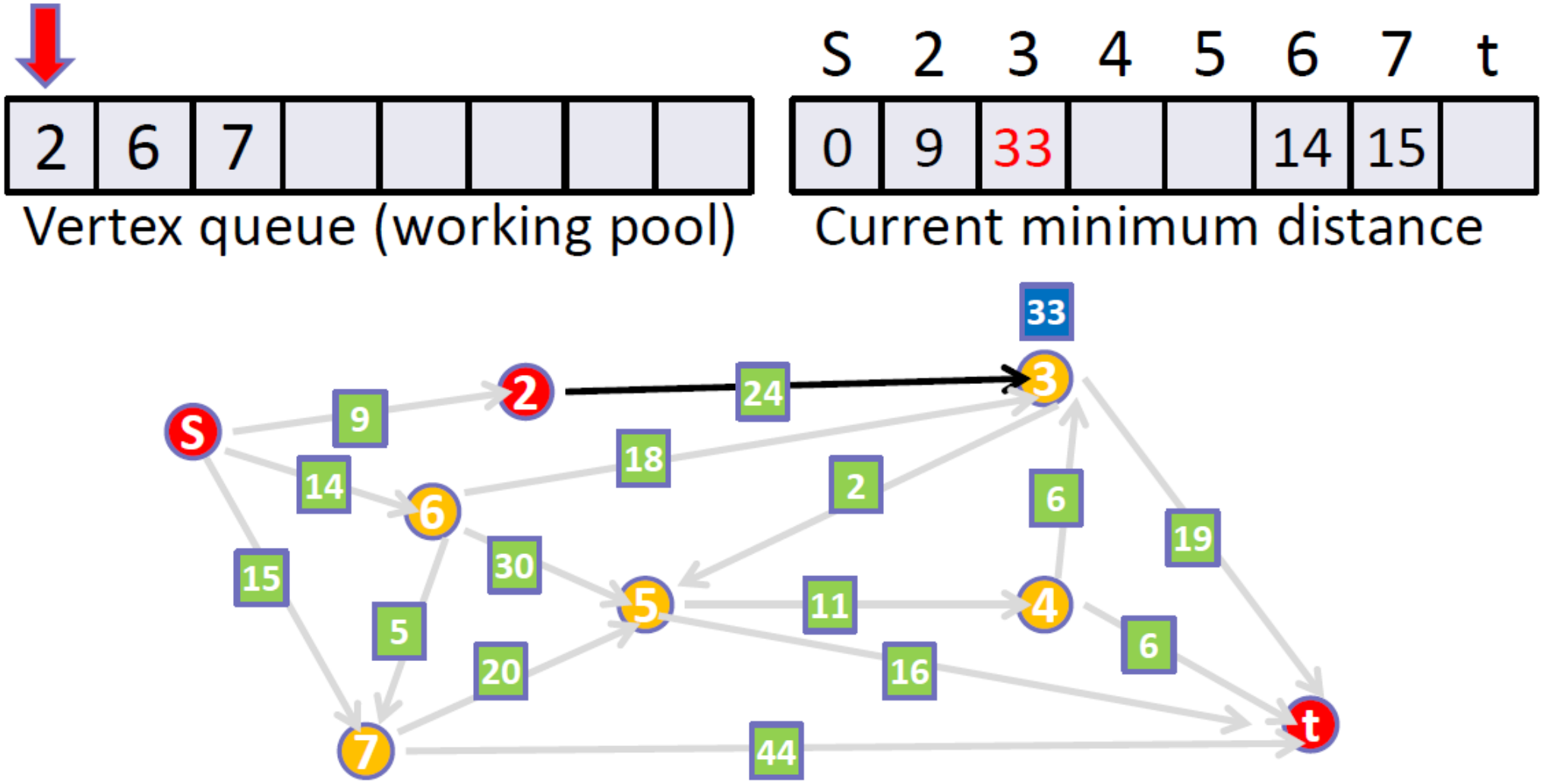
- Let d_i be the shortest distance from source to v_i
- Let $w_{i,j}$ be the weight between v_i and v_j
- Iteratively search over the graph
  - If d_i be the new shortest distance from source to v_i
  - For each v_j adjacent to v_i, update the shortest distance to v_j

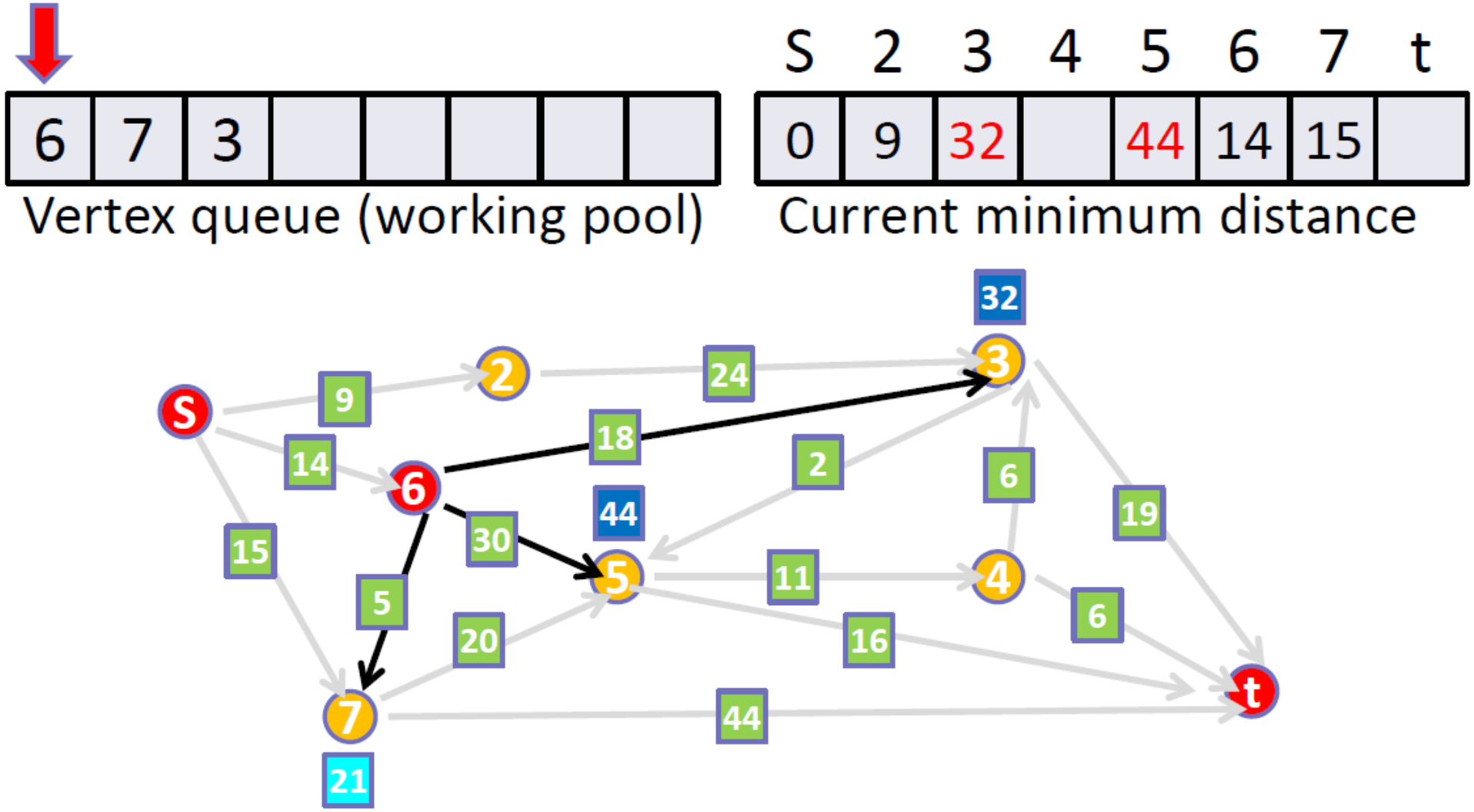
$$new\_d_j = \min(d_j, d_i + w_{i,j})$$

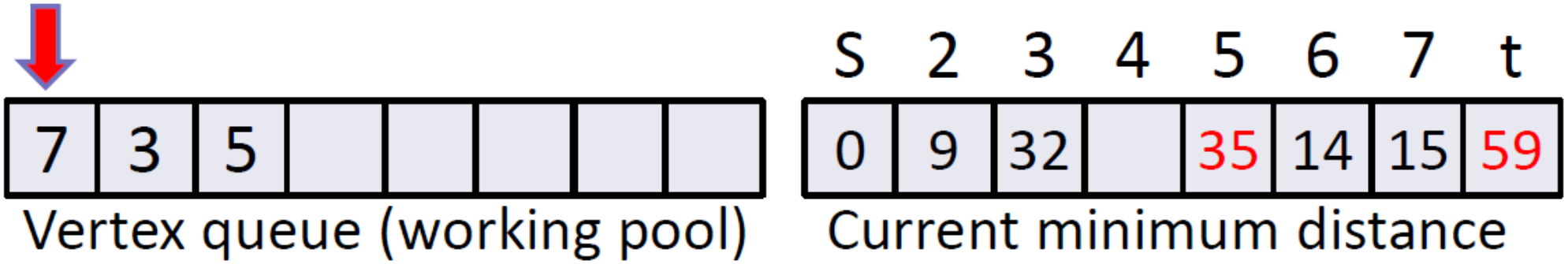


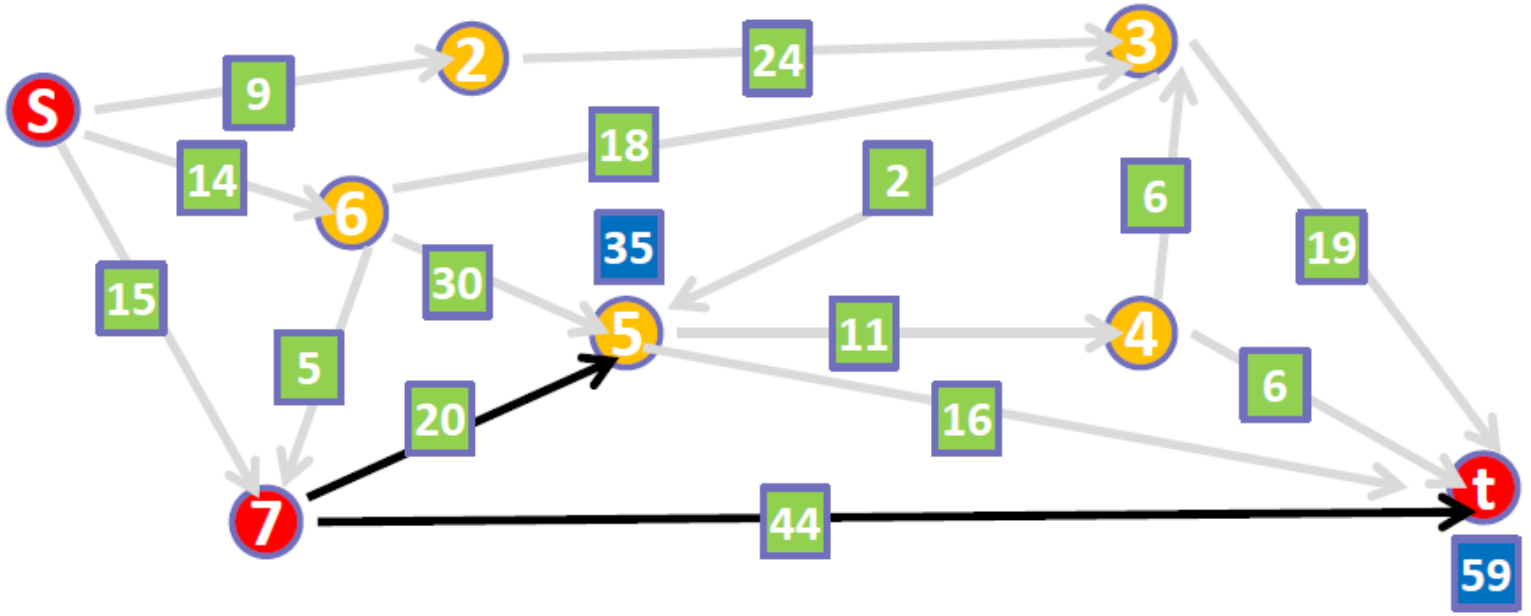

S
Vertex queue (working pool)
S 2 3 4 5 6 7 t
0 9 14 15
Current minimum distance

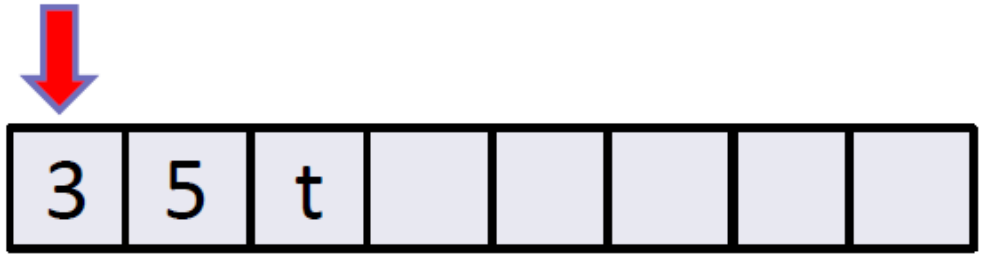

9
S
9
2
24
3
14
14
18
6
2
6
19
15
30
5
11
4
5
6
20
16
7
44
t
15


S 2 3 4 5 6 7 t
2 6 7
0 9 33 14 15
Vertex queue (working pool)
Current minimum distance
33
S
2
3
24
9
18
14
6
2
6
19
30
15
5
11
4
5
6
20
16
7
44
t


S 2 3 4 5 6 7 t
6 7 3
0 9 32 44 14 15
Vertex queue (working pool)
Current minimum distance
32
S
9
2
24
3
18
14
6
2
6
44
19
30
15
5
11
4
5
6
20
16
t
7
44
21

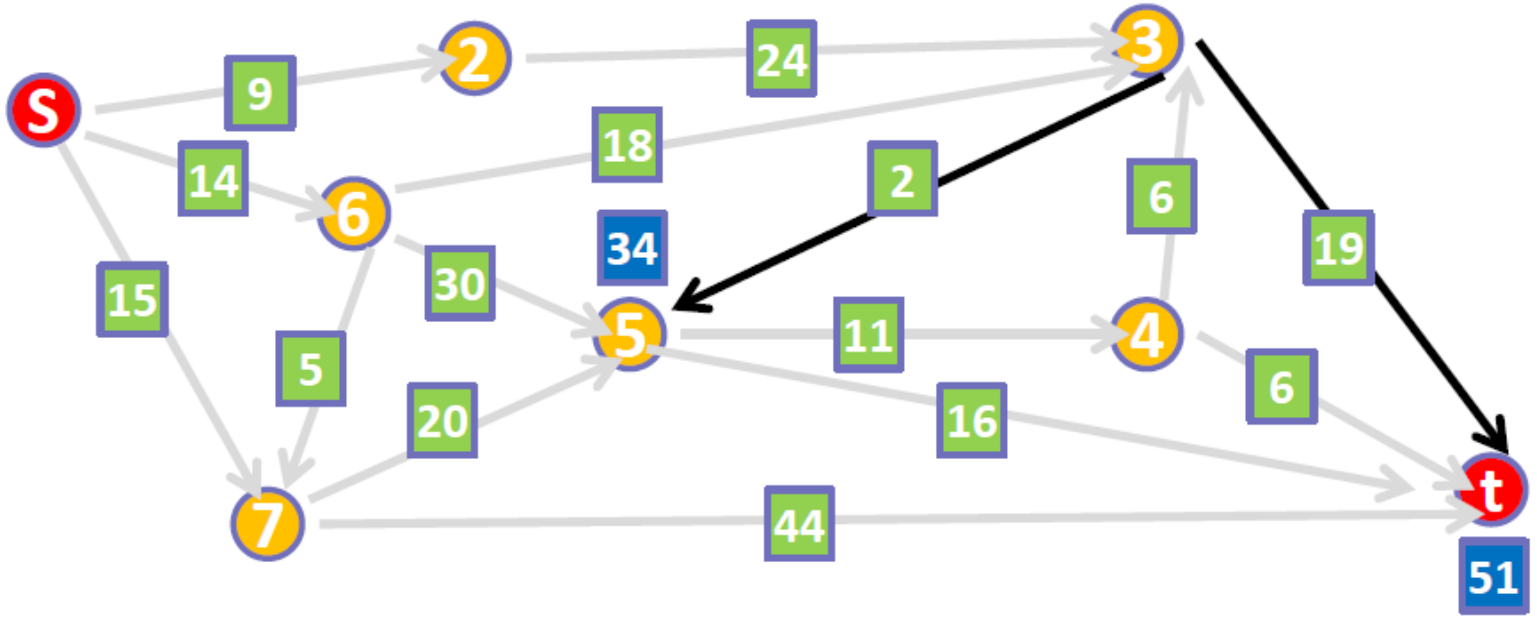

7 3 5
Vertex queue (working pool)
S 2 3 4 5 6 7 t
0 9 32 35 14 15 59
Current minimum distance

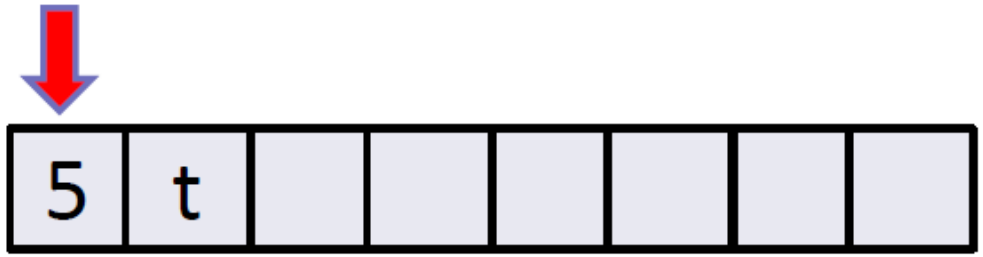

S
2
3
6
5
4
7
t
9
24
14
18
2
6
19
35
30
15
11
5
6
20
16
44
59



S	2	3	4	5	6	7	t
0	9	32		34	14	15	51

Current minimum distance

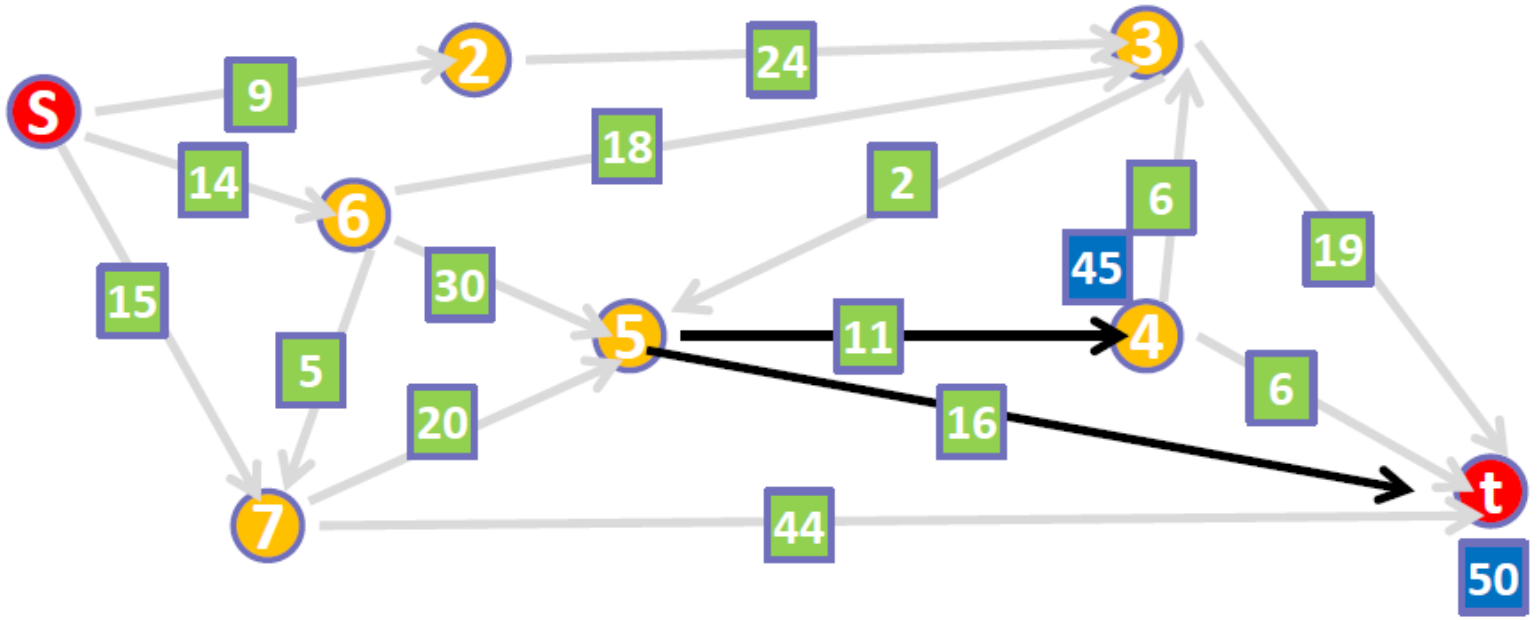


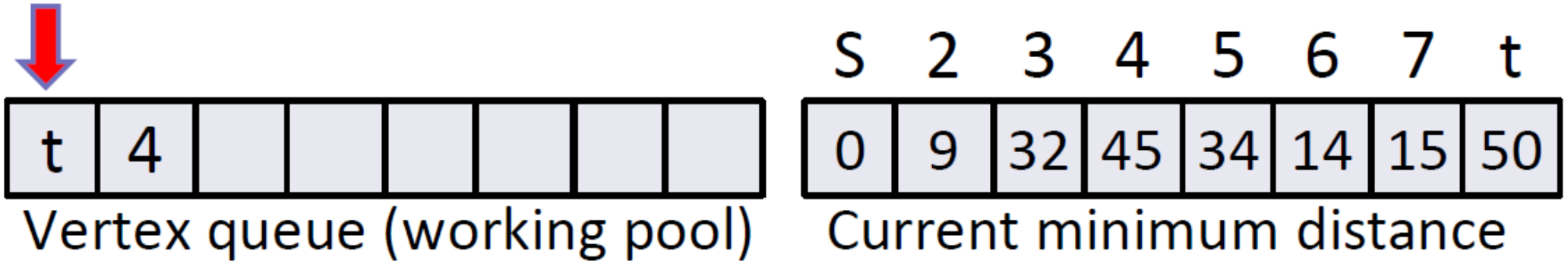


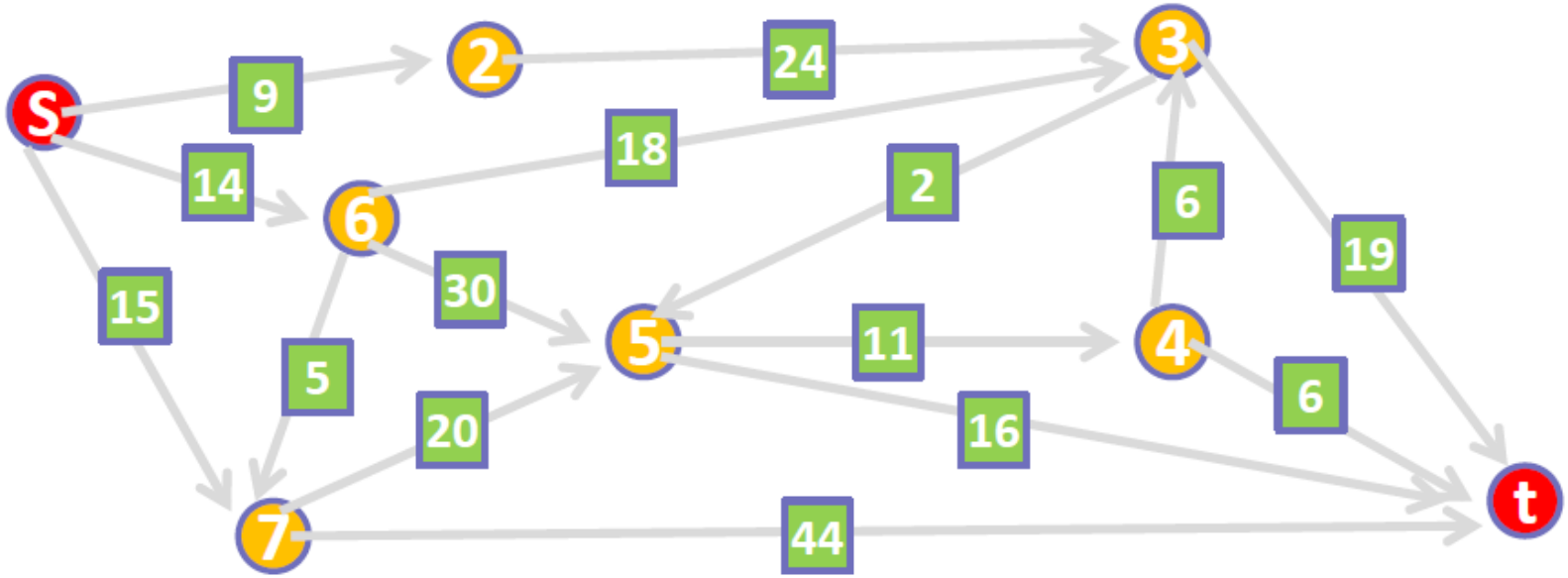
Vertex queue (working pool)

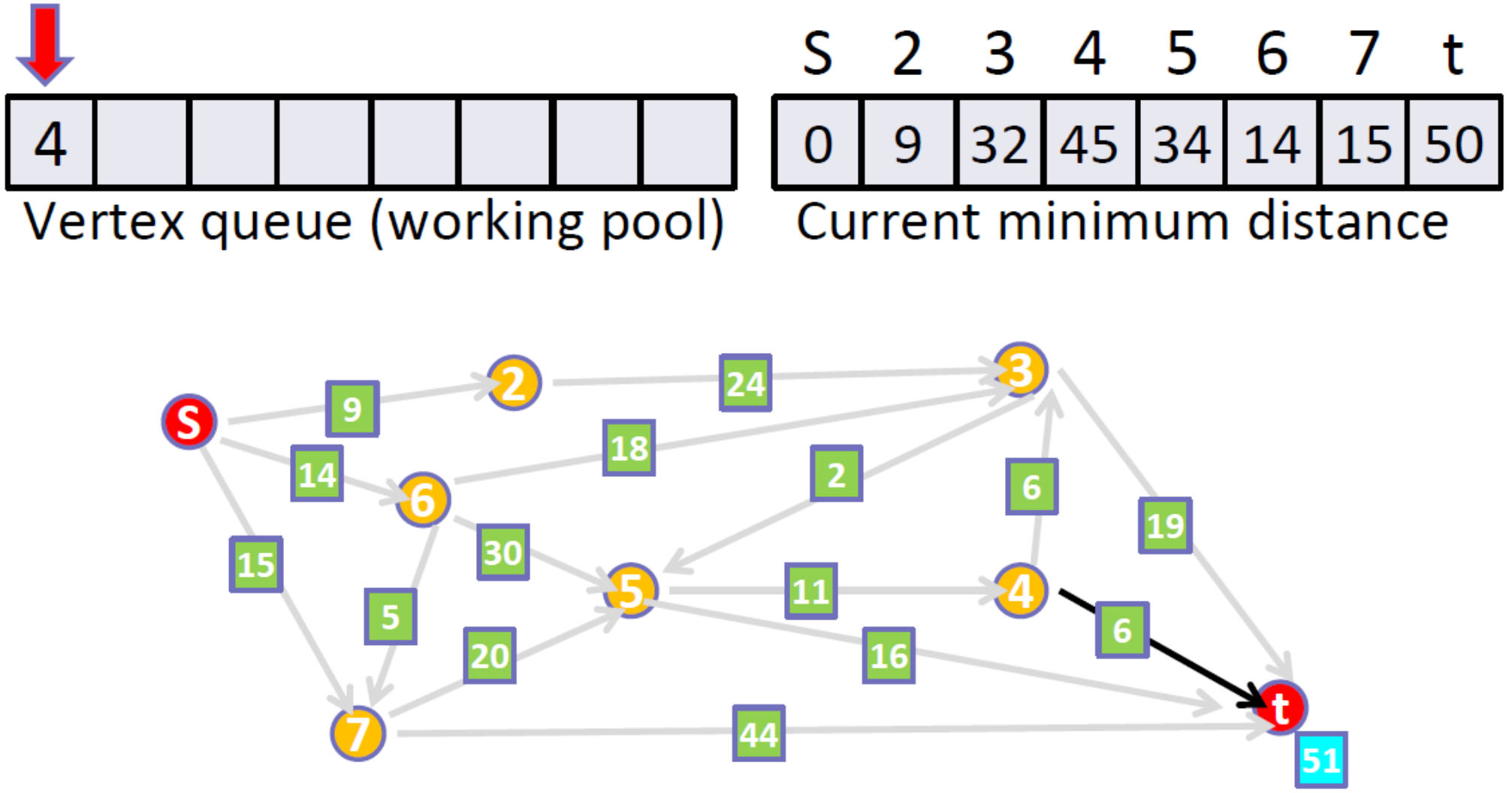
S	2	3	4	5	6	7	t
0	9	32	45	34	14	15	50

Current minimum distance




t
4
Vertex queue (working pool)
S 2 3 4 5 6 7 t
0 9 32 45 34 14 15 50
Current minimum distance

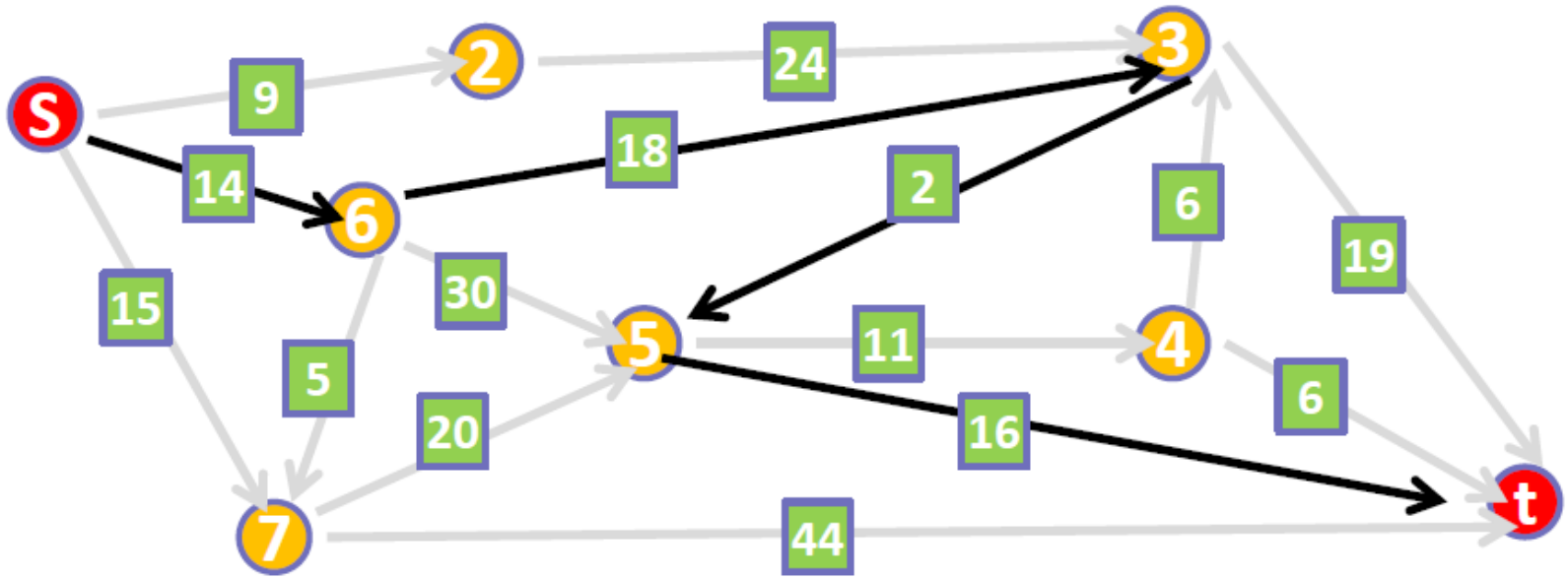

S
2
3
4
5
6
7
t
9
14
24
18
2
6
19
15
30
11
5
20
16
6
44


4
Vertex queue (working pool)
S 2 3 4 5 6 7 t
0 9 32 45 34 14 15 50
Current minimum distance
S
2
3
4
5
6
7
t
9
24
14
18
2
6
19
15
30
11
5
6
20
16
44
51

Vertex queue (working pool)

S	2	3	4	5	6	7	t
0	9	32	45	34	14	15	50

Current minimum distance

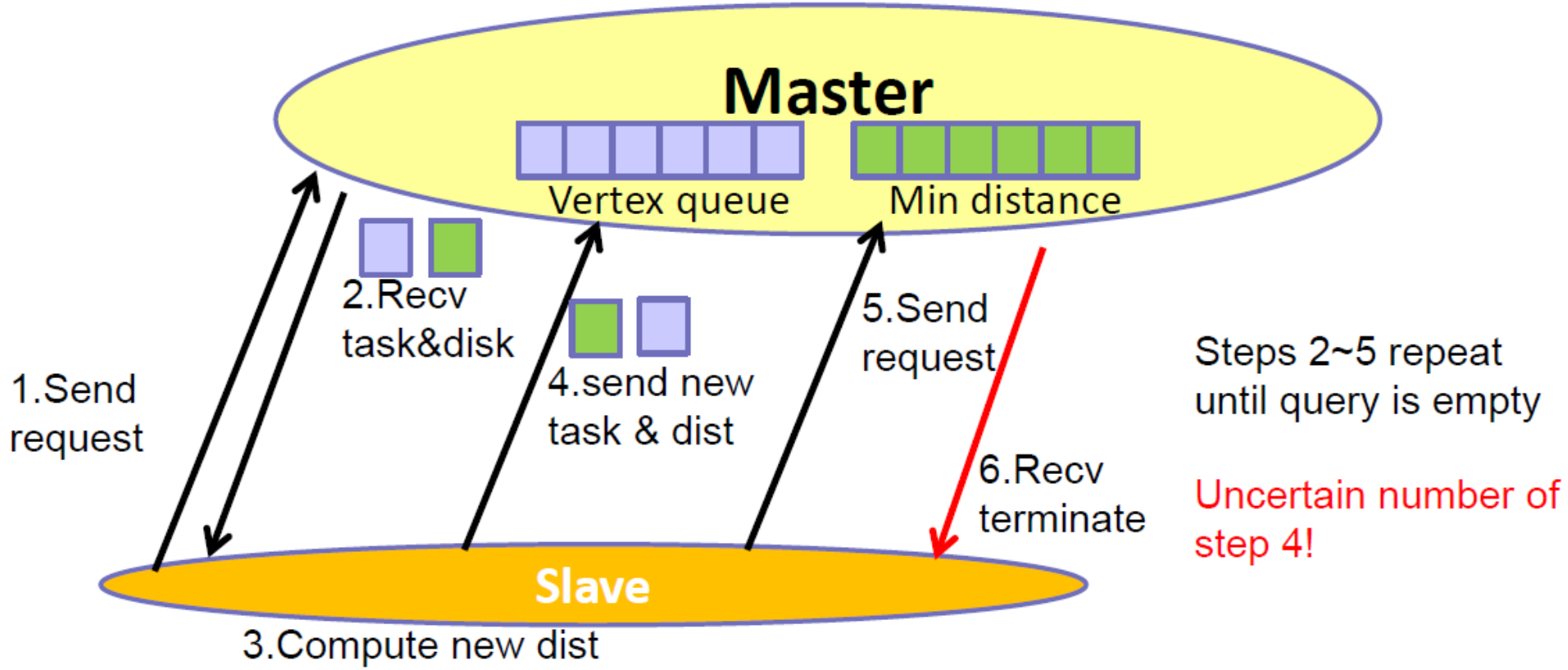


Sequential Code

```
while ((i = next_vertex()) != no_vertex)      /* while a vertex */
   for (j = 0; j < n; j++) {                  /* get next edge */
         if (w[i][j] != infinity) {           /* if an edge */
               newdist_j = dist[i] + w[i][j];
               if (newdist_j < dist[j]) {
                  dist[j] = newdist_j;
                  enqueue(j);    /* enqueue vertex if not there */
               }
         }
   }
} /* no more vertices to consider */
```

Centralized Pool Parallel Code

- Centralized work pool holds the vertex queue and distance array
- Each slave takes a vertex and returns new vertices and distances
- Since graph weights is fixed, it could be copied into each slave.



Centralized Pool Parallel Code

- Master

```
while (vertex_queue() != empty) {
      recv(PANY, source = Pi);          /* request task from slave */
      v = get_vertex_queue();
      send(&v, Pi);                     /* send next vertex and */
      send(&dist, &n, Pi);              /* current dist array */

      n_recv(&count, PANY, source = Pi);        /* nonblocking call to wait slave*/
      // once recv do the following call
      for (j=0; j<count; j++) {
            recv(&j, &dist[j], PANY, source = Pi);   /* new distance */
            append_queue(j, dist[j]);   /* append vertex to queue */
                                                /* and update distance array */
      }
};
recv(PANY, source = Pi);                /* request task from slave */
send(Pi, termination_tag);              /* termination message*/
```

Slave

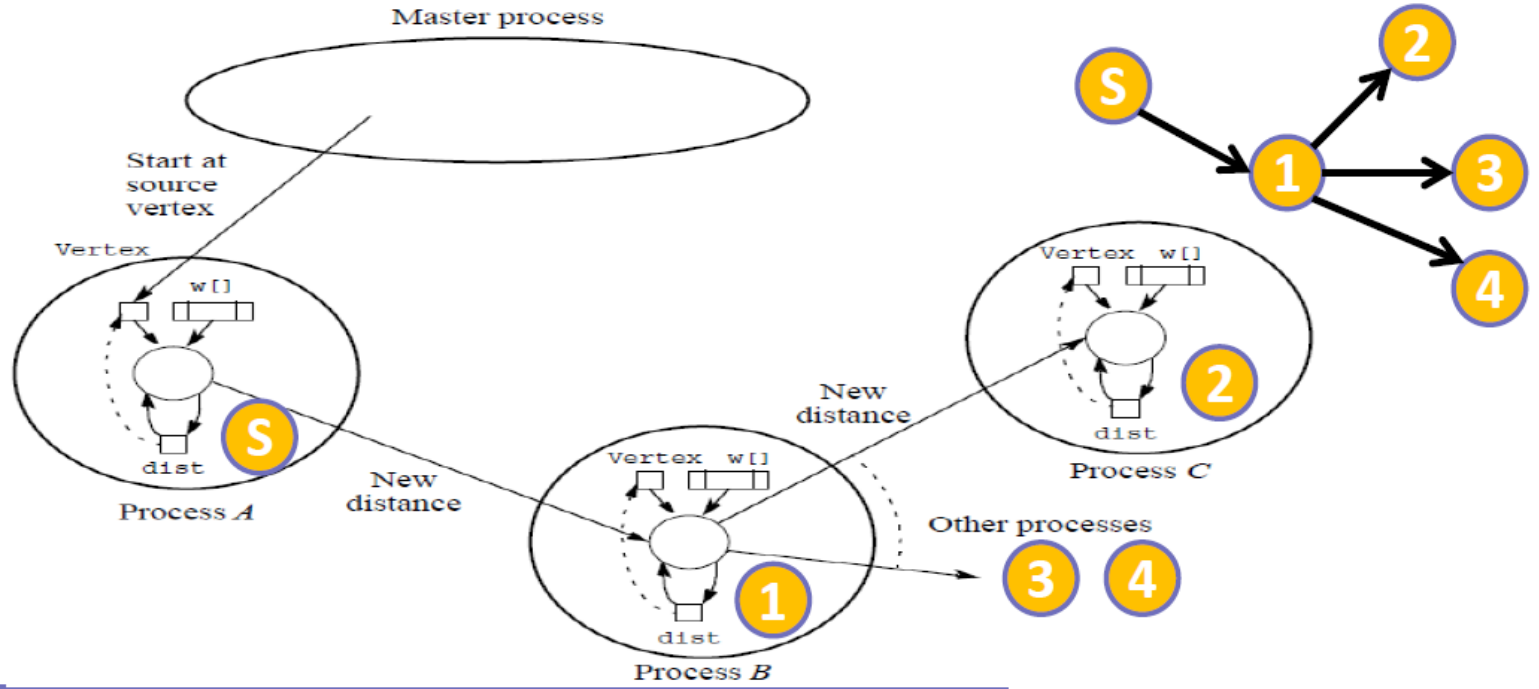
```
send(Pmaster);                          /* send request for task */
recv(&v, Pmaster, tag);                 /* get vertex number */
while (tag != termination_tag) {
      recv(&dist, &n, Pmaster); /* and dist array */
      count=0;
      for (j = 1; j < n; j++) {         /* get next edge */
            if (w[v][j] != infinity) {  /* if an edge */
                  newdist_j = dist[v] + w[v][j];
                  if (newdist_j < dist[j]) {
                        new_vertex[count] = j;
                        new_dist[count] = newdist_j;
                        count++;
                  }
            }
      }
      send(&count, Pmaster)   /* the number of new vertex*/
      for (j = 1; j < n; j++)           /* send each new distance and vertex*/
            send(&new_vertex[j], &new_dist[j], Pmaster);
       send(Pmaster);                                 /* send request for task */
       recv(&v, Pmaster, tag);          /* get vertex number */
}
```

Distributed Pool Parallel Code

- Distribute vertex queue and distance array
  - Each process holds all the information of a vertex
  - Including queue, distance and adjacent weight

- Algorithm
  - For each new distance to a vertex V_i is computed
    - → send the distance to P_i and re-activate P_i
    - → if the new distance is shorter:
      - compute new distance for each adjacent vertex V_j
      - send the new distance to P_j

Distributed Pool Parallel Code




```
recv(newdist, PANY);
if (newdist < dist)
      dist = newdist; /* start searching around vertex */
      for (j = 1; j < n; j++) /* get next edge */
            if (w[j] != infinity) {
                  d = dist + w[j];
                  send(&d, Pj); /* send distance to proc j */
            }
```

Q: How do we know when to terminate?

A: Use dual-pass ring algorithm…

Dual-Pass Ring Termination Algorithm

1. *P*0 becomes white when it has terminated and generates a white token to *P*1.
2. The token is passed through the ring from one process to the next when each process has terminated, but the color of the token may be changed.
   - If *Pi* passes a task to *Pj* where $j < i$ (that is, before this process in the ring), it becomes a *black process*; otherwise it is a *white process*.
   - A black process will color a token black and pass it on.
   - A white process will pass on the token in its original color (either black or white). After *Pi* has passed on a token, it becomes a white process. *Pn*-1 passes the token to *P*0.
3. When *P*0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

```
Master process (P0)
for (i = 0; i < no_tasks; i++) {
    recv(P1, request_tag);          /* request for task */
    send(&task, Pi, task_tag);      /* send tasks into queue */
}
recv(P1, request_tag);              /* request for task */
send(&empty, Pi, task_tag);         /* end of tasks */
```

```
Process Pi (1 < i < n)
if (buffer == empty) {
    send(Pi-1, request_tag);        /* request new task */
    recv(&buffer, Pi-1, task_tag);  /* task from left proc */
}
if ((buffer == full) && (!busy)) {  /* get next task */
    task = buffer;                  /* get task*/
    buffer = empty;                 /* set buffer empty */
    busy = TRUE;                    /* set process busy */
}
nrecv(Pi+1, request_tag, request); /* check msg from right */
if (request && (buffer == full)) {
    send(&buffer, Pi+1);            /* shift task forward */
    buffer = empty;
}
if (busy) {                         /* continue on current task */
    Do some work on task.
    If task finished, set busy to false.
}
```

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

稀疏矩阵乘法

稀疏矩阵是大部分元素为 0 的矩阵。很多科学、工程、金融建模问题都要用到稀疏矩阵。例如，矩阵经常用来表示线性方程组的系数，正如我们在第 7 章中看到的，每一行矩阵代表一个方程。很多科学和工程问题中有大量的变量，包含这些变量的方程是松耦合的，也就是说每个方程中只含有几个变量。图 10-1 说明了这一点，方程 0 中含有变量 x_0 和 x_2，方程 1 中没有变量，变量 x_1、x_2 和 x_3 在方程 2 中，最后变量 x_0 和 x_3 在方程 3 中。

第 0 行	3	0	1	0
第 1 行	0	0	0	0
第 2 行	0	2	4	1
第 3 行	1	0	0	1

图 10-1　稀疏矩阵的简单例子

保存稀疏矩阵时不保存 0 元素。我们首先介绍的是稀疏行压缩(Compressed Sparse Row，CSR)存储格式，如图 10-2 所示。CSR 只在一维数据中保存非 0 值，即图 10-2 中的 data[]。数组 data[]要保存图 10-1 中稀疏矩阵中的所有非 0 值，为此它需要先保存第 0 行的非 0 元素(3 和 1)，然后保存第 1 行的非 0 元素(没有)，再保存第 2 行的非 0 元素(2、4 和 1)，最后保存第 3 行的非 0 元素(1 和 1)。这种格式把所有 0 元素都给压缩掉了。

	第 0 行	第 2 行	第 3 行
非 0 值 data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
列索引 col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
行指针 row_ptr[5]	{ 0, 2, 2, 5, 7 }		

图 10-2　CSR 格式的示例

为了这样压缩格式，我们需要引入两组标记来保存原来的稀疏矩阵结构。第一组标记构成了列索引矩阵，如图 10-2 中的 col_index[]所示，它给出了每个非 0 值在原稀疏矩阵中的列索引。因为我们把每一行中的 0 元素给压缩掉了，所以需要用这组标记来记住非 0 元素在原稀疏矩阵该行中的位置。例如，3 和 1 这两个值原来在稀疏矩阵第 0 行的第 0 列和第 2 列，我们用 col_index[0]和 col_index[1]来保存这两个元素的列索引。又如，2、4 和 1 这三个值原来在稀疏矩阵的第 1、2 和 3 列，所以 col_index[0]、col_index[1]和 col_index[2]保存了索引 1、2 和 3。

值。由于稀疏矩阵-向量乘加的重要性，人们创建了一个叫 SpMV(sparse matrix-vector multiplication，稀疏矩阵-向量乘法)的标准库函数接口，我们将用 SpMV 来解释稀疏计算的不同存储格式之间的重要权衡。

(1)num_rows，这是一个函数参数，它规定了稀疏矩阵的行数；(2)浮点数组 data[]、两个整型数组 row_ptr[]和 x[]，如图 10-3 所示。代码只有 7 行，第 1 行是一个循环，它循环访问矩阵的每一行，在每次迭代中计算当前这一行和向量 x 的点积。

```
1.   for (introw = 0; row < num_rows; row++) {
2.     float dot = 0;
3.     int row_start = row_ptr[row];
4.     int row_end =   row_ptr[row+1];

5.     for (intelem = row_start; elem < row_end; elem++) {
6.         dot += data[elem] * x[col_index[elem]];
       }

7.     y[row] += dot;

     }
```

图 10-4　实现 SpMV 的串行循环

在每一行中，第 2 行代码首先将点积初始化为 0，然后设置属于这一行的 data[]数组元素的范围。起始位置和结束位置可以从 row_ptr[]数组中读取。图 10-5 以图 10-1 中小稀疏矩阵为例解释了这个过程。在 row=0 时，row_ptr[row]是 0，row_ptr[row+1]是 2。注意，这两个来自第 0 行的元素保存在 data[0]和 data[1]中，也就是说，row_ptr[row]给出了当前行的起始位置，row_ptr[row+1]给出了下一行的起始位置，也就是当前这一行的结束位置后面的那个位置。第 5 行中的循环反映了这一点，循环索引将从 row_ptr[row]迭代到 row_ptr[row+1]-1。

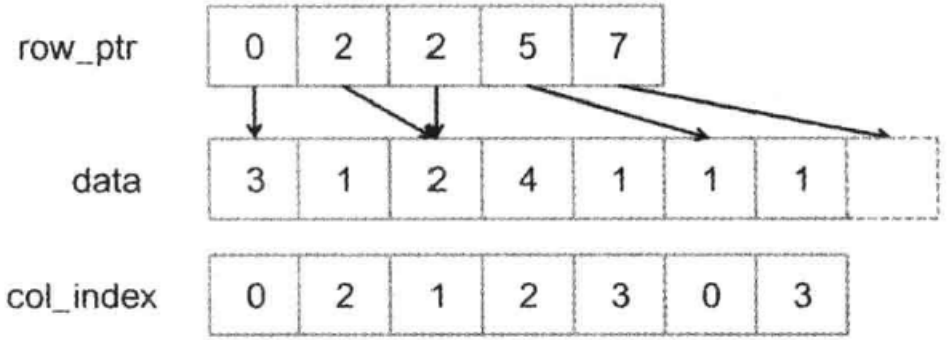


图 10-5　对图 10-1 中的稀疏矩阵使用 SpMV 循环

第 6 行中的循环体计算了当前这行的点积。对每个元素，它用循环索引 elem 来访问矩阵元素 data[elem]，同时还用 elem 从 col_index[elem]提取元素的列索引，然后我们用这个列索引来访问乘法所需要的 x 元素。比如，data[0]中的元素来自第 0 列(col_index[0]=0)，data[1]中的元素来自第 2 列(col_index[1]=2)。因此，内层循环将通过 data[0]*x[0]+data[1]*x[2]计算出第 0 行的点积。读者可以计算出其他几行的点积，我们将它留作习题。

CSR 彻底消除了存储中的所有 0 元素，但它引入了 col_index 和 row_ptr 数组，因此带来了额外开销。在我们的小例子中，0 元素的数量并不比非 0 元素多太多，额外开销比要

10.2 使用 CSR 格式的并行 SpMV

需要注意的是，稀疏矩阵每一行的点积运算都与其他行无关，这反映在图 10-4 中所有外层循环(第 1 行)的迭代在逻辑上互相独立。通过把外层循环的每次迭代分配给一个线程，我们很容易把串行 SpMV/CSR 转化为并行的 CUDA kernel，如图 10-6 所示，其中线程 0 计算第 0 行的点积，线程 1 计算第 1 行的点积，以此类推。

线程0	3	0	1	0
线程1	0	0	0	0
线程2	0	2	4	1
线程3	1	0	0	1

图 10-6 将线程映射到并行 SpMV/CSR 中的行的示例

真实的稀疏矩阵运算通常有成千上万行要计算，每行中都包含大量非 0 元素。这样看来图 10-6 中的映射方法就很合适：有很多线程，每个线程都要做大量工作。我们在图 10-7 中给出了一个并行版的 SpMV/CSR。

真实的稀疏矩阵运算通常有成千上万行要计算，每行中都包含大量非 0 元素。这样看来图 10-6 中的映射方法就很合适：有很多线程，每个线程都要做大量工作。我们在图 10-7 中给出了一个并行版的 SpMV/CSR。

```
1.  __global__ void SpMV_CSR(int num_rows, float *data, int *col_index,
      int *row_ptr, float *x, float *y) {

2.    int row = blockIdx.x * blockDim.x + threadIdx.x;

3.    if (row < num_rows) {
4.      float dot = 0;
5.      int row_start = row_ptr[row];
6.      int row_end =   row_ptr[row+1];
7.      for (int elem = row_start; elem < row_end; elem++) {
8.        dot += data[elem] * x[col_index[elem]];
        }
9.      y[row] = dot;
      }

    }
```

图 10-7　并行 SpMV/CSR kernel

虽然并行 SpMV/CSR kernel 很简单，但它有两大缺点。第一个缺点是它不能合并访存，如果读者检查一下图 10-5，就会发现相邻线程会同时访问不相邻的存储器位置。在我们的小例子中，在点积循环的第一次迭代中，线程 0 将访问 data[0]，线程 1 跳过，线程 2 将访问 data[2]，线程 3 将访问 data[5]。显然，这些相邻的线程没有同时访问相邻的存储器位置，因此，并行 SpMV/CSR kernel 无法有效利用存储器带宽。

SpMV/CSR kernel 的第二个缺点是所有 warp 都有很严重的控制流分支。一个线程点积循环的迭代次数取决于分配给线程的那一行中非 0 元素的个数。因为非 0 元素随机分布在各行中间，所以相邻两行的非 0 元素数目可能相差很大，这就导致了绝大多数 warp 都可能会发生控制流分支，甚至所有 warp 都可能会发生控制流分支。

可以明显地看到，并行 SpMV kernel 的执行效率和存储器带宽利用率都和输入数据矩阵的分布有关，这和我们之前介绍过的绝大多数 kernel 都不同，然而，现实世界很多应用都具有这种性能行为依赖于数据的特点，这就是为什么 SpMV 是很重要的并行模式的原因——虽然它很简单，但它揭示了很多复杂并行程序所共有的重要行为。我们将在下一节中讨论几种解决并行 SpMV/CSR kernel 缺点的重要技术。

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

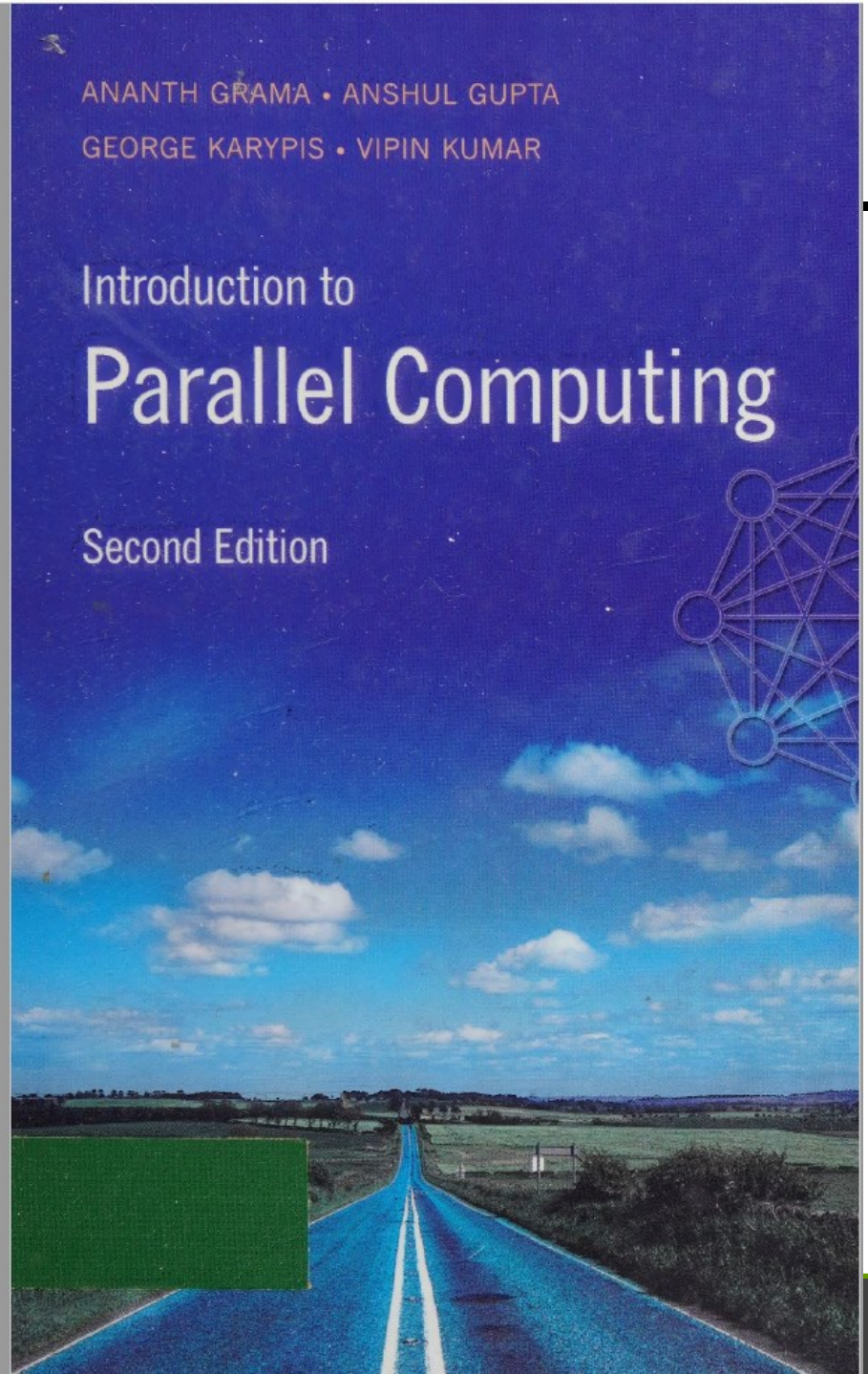
第13章　快速傅里叶变换

离散傅里叶变换（DFT）在许多科学和技术应用中发挥着重要的作用，如时间序列和波形分析、线性偏微分方程求解、卷积、数字信号处理以及图像滤波等。离散傅里叶变换是一种线性变换，它把某个周期性信号（如正弦波）单个周期中的n个规则抽样点映射到相同数目的点上，这些点表示这个信号的频谱。1965年，Cooley和Tukey提出一个运算量为$\Theta(n \log n)$的算法，计算任一n点序列的离散傅里叶变换。与过去已知的运算量为$\Theta(n^2)$的其他计算离散傅里叶变换的算法比较，他们的新算法有了显著的改进。Cooley和Tukey的这一革命性的算法及其变体称为*快速傅里叶变换*（Fast Fourier Transform, FFT）。由于它在科学与工程领域的广泛运用，在并行计算机上实现快速傅里叶变换一直倍受关注。

快速傅里叶变换算法有几种不同的形式。本章讨论它的最简形式：一维的、无序的和基数为2的快速傅里叶变换。更高基数的、多维的快速傅里叶变换的并行形式与本章讨论的简单算法类似，因为所有串行快速傅里叶变换的基本思想是一样的。有序的快速傅里叶变换通过对无序快速傅里叶变换的输出序列使用位反转（13.4节）得到。位反转并不影响快速傅里叶变换并行实现的整体复杂度。

书签

- Contents
- Preface
- 1 Introduction to Parallel Computing
- 2 Parallel Programming Platforms
- 3 Principles of Parallel Algorithm Design
- 4 Basic Communication Operations
- 5 Analytical Modeling of Parallel Programs
- 6 Programming Using the Message-Passing Paradigm
- 7 Programming Shared Address Space Platforms
- 8 Dense Matrix Algorithms
- 9 Sorting
- 10 Graph Algorithms
- 11 Search Algorithms for Discrete Optimization Problems
- 12 Dynamic Programming
- 13 Fast Fourier Transform
- A Complexity of Functions and Order Analysis
- Bibliography

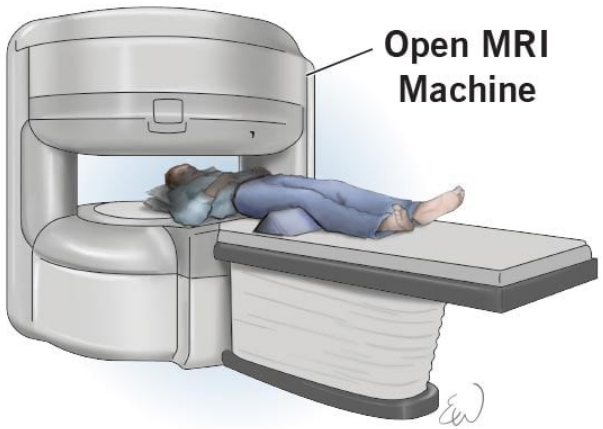


10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

11.1 应用背景

MRI



Cleveland Clinic ©2022

读者要想了解 MRI 的物理学原理，可以参阅 MRI 手册，如 Liang 和 Lauterbur[LL1999]。在这个案例的学习过程中，我们主要关注在重构阶段中的计算复杂性，以及 k-空间采样轨迹如何影响这种复杂性。MRI 扫描仪用到的 k-空间采样轨迹很大程度上会影响重构图像的质量、重构算法的时间复杂度，以及扫描仪采集样本需要的时间。在一类重构方法中，k-空间样本和重构图像之间的关系如公式(11.1)所示。

$$\hat{m}(\mathrm{r}) = \sum_{j} W(\mathrm{k}_j) s(\mathrm{k}_j) e^{i2\pi \mathrm{k}_j \cdot \mathrm{r}} \tag{11.1}$$

在公式(11.1)中，m(r)是重建图像，s(k)是已测量的 k-空间的数据，而 W(k)是用于非均匀采样的加权函数。也就是说，W(k)可以用于减少 k-空间区域内高密度采样点带来的影响。对于这类重构，W(k)也可以用作切趾函数(apodization function)(也叫窗函数(window function)——译者注)，来减少噪声带来的影响，并减少由于样本有限而产生的伪影。

如果在 k-空间中采集的数据在理想条件下在等间隔的笛卡尔空间内的网格点上进行，那么 W(k)这个加权函数是一个常数，因此这个常数可以作为公因数从公式(11.1)的求和式中提取出来。因此，对 m(r)的重构实际上变成求 s(k)的快速傅立叶变换(Fast Fourier

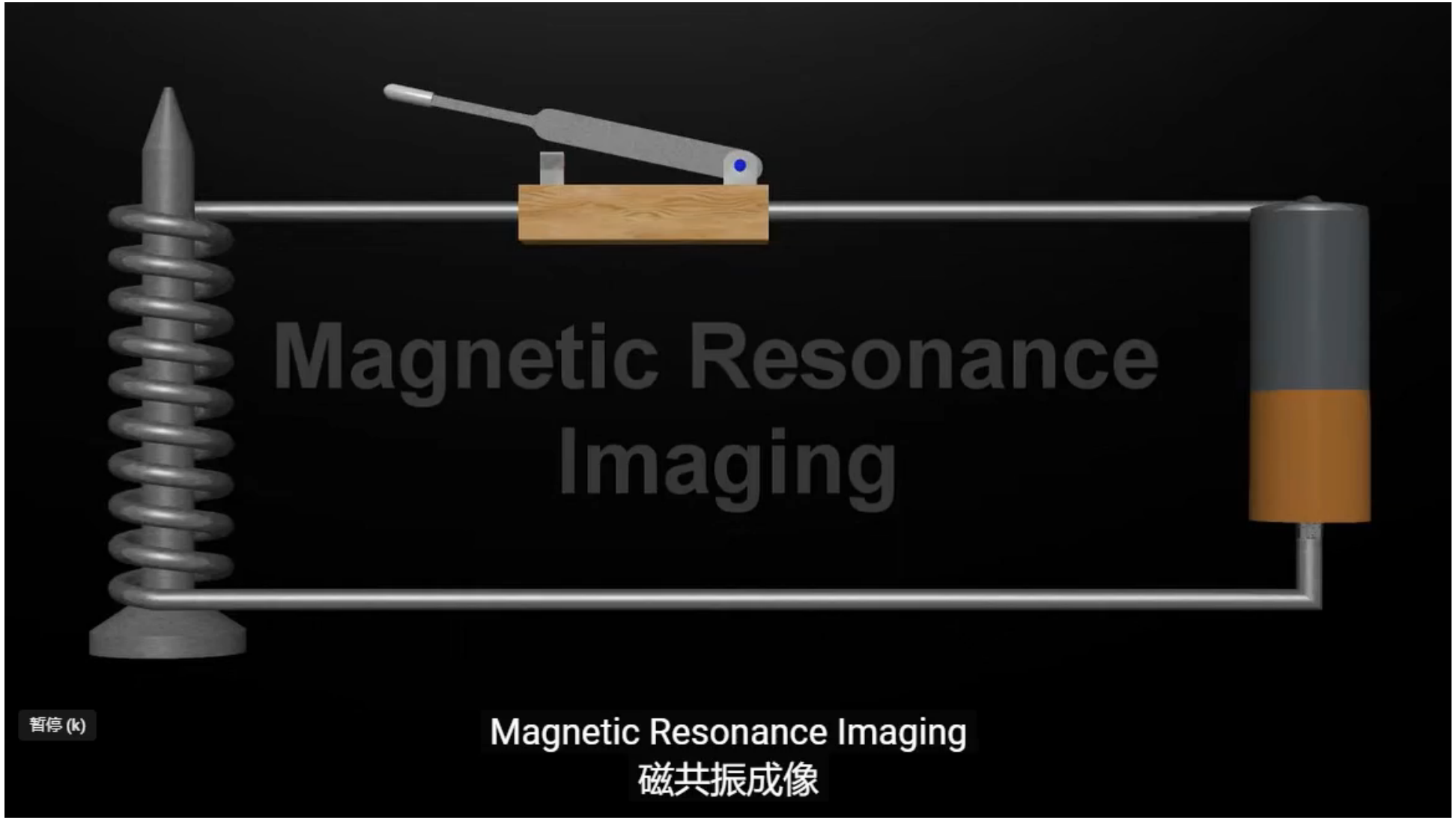
Transform，FFT)的逆变换，这是一种很有效的计算方法。在等间隔的笛卡尔空间内网格点上的数据的集合也称为笛卡尔扫描轨迹(Cartesian scan trajectory)。图 11-1(a)描述了笛卡尔扫描轨迹。实际上，笛卡尔扫描轨迹可以直接在扫描仪上实现，并广泛地应用于临床环境中。

第 1 步：确定 kernel 函数的并行结构

从概念上来讲，将循环转化成CUDA 中的 kernel 函数是一件很简单的事。因为图 11-4(b) 外层循环中的所有迭代过程都可以并行执行，所以我们在把外层循环转化成 CUDA 中的 kernel 函数时，只需要把迭代过程映射到 CUDA 线程。图 11-5 展示了通过这种简单的转化得到的 kernel 函数。每个线程实现原来外层循环中的一次迭代过程，也就是说，我们用每个线程来计算 k-空间中的一个样本对 F^HD 中的所有元素所产生的影响。原始外层循环中有 M 次迭代，而 M 可能是几百万。显然，我们需要多个线程块来产生足够多的线程以实现所有的迭代过程。

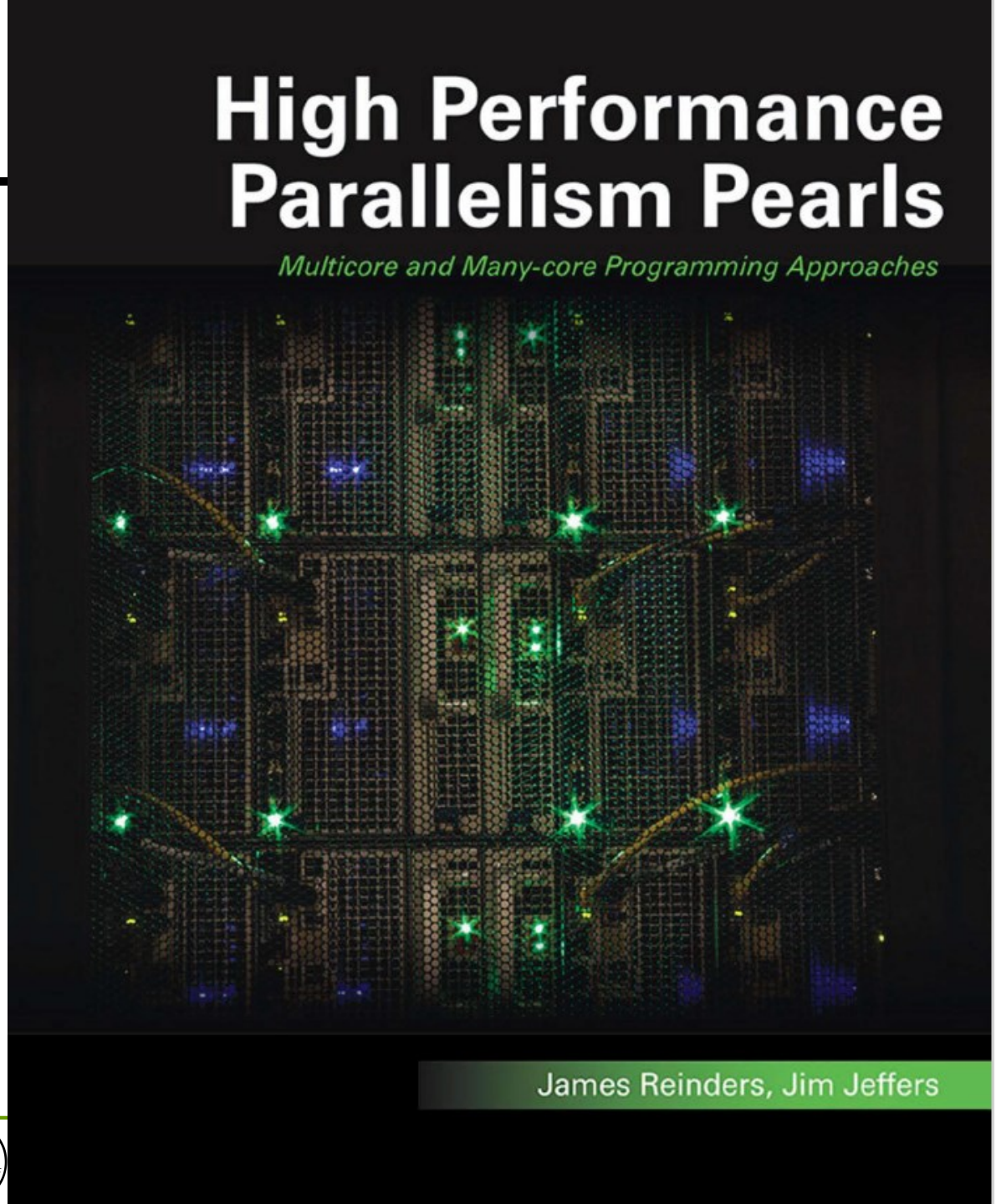
```
__global__ void cmpFhD(float* rPhi, iPhi, rD, iD,
    kx, ky, kz, x, y, z, rMu, iMu, rFhD, iFhD, int N) {

  int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

  rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
  iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];

  for (n = 0; n < N; n++) {
    floatexpFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);

    floatcArg = cos(expFhD);  floatsArg = sin(expFhD);

    rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
    iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
  }
}
```

图 11-5 这是 F^HD kernel 函数的第一个版本。由于线程在写入 rFhD 和 iFhD 数组时会发生冲突，因


Magnetic Resonance
Imaging
暂停 (k)
Magnetic Resonance Imaging
磁共振成像

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …



High Performance
Parallelism Pearls
Multicore and Many-core
Programming Approaches

N-BODY SIMULATIONS

N-Body simulations are a common astrophysics problem that computes the movement of gravitationally interacting particles through space. At each time step of the simulation, the force that each particle exacts upon each other is computed using the following equation:

$$F_{ij} = K \frac{m_1 m_2 (\vec{r_i} - \vec{r_j})}{|\vec{r_i} - \vec{r_j}|^3}$$

Based on this, the velocity and position of each particle are updated. The process is iterative and finishes after simulation of the desired number of time steps.

While the direct N-body kernel is generally not practical by itself because of its $O(n^2)$ complexity, it is still used to compute the interaction of particles of the same domain in more complex N-body algorithms which have smaller computational complexity, like Barnes-Hut that has an $O(n \log n)$ complexity, and in some scenarios where the numerical approximations of other methods are not desirable such as the study of the evolution of star clusters.

Figure 9.2 shows the time step of the N-body simulation where on each time step the above kernel is called and then the positions of the different particles are updated.

The hardware used for the evaluation described in this chapter was an Intel Xeon Phi coprocessor 7120P (with turbo disabled unless otherwise noted). It features 61 cores at 1.23 GHz for a total of 244

```
template <class T>
void computeForces(ParticleSystem<T> p, const T dt)
{
  int n = p.nParticles;
  #pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < n; i++) {
    T Fx = static_cast<T>(0.0);
    T Fy = static_cast<T>(0.0);
    T Fz = static_cast<T>(0.0);
    for (int j = 0; j < n; j++) {
        const T dx = p.x[j] - p.x[i];
        const T dy = p.y[j] - p.y[i];
        const T dz = p.z[j] - p.z[i];

        const T drSquared = dx*dx + dy*dy + dz*dz
                          + p.softening;
        const T drPowerN12 = 1.0f / sqrtf(drSquared);
        const T drPowerN32 = drPowerN12 * drPowerN12 *
                             drPowerN12;
        const T s = p.m[j] * drPowerN32;

        Fx += dx * s; Fy += dy * s; Fz += dz * s;
    }
    p.vx[i] += dt*Fx; p.vy[i] += dt*Fy; p.vz[i] += dt*Fz;
  }
}
```

FIGURE 9.1

Initial N-body kernel.

```
for (int iter = 0; iter < nIters; iter++)    {
   computeForces(p,dt);
   for (int i = 0; i < p.nParticles; i++) {
       p.x[i] += p.vx[i] * dt;
       p.y[i] += p.vy[i] * dt;
       p.z[i] += p.vz[i] * dt;
   }
}
```

FIGURE 9.2

Time step loop.

使用四阶龙格库塔方法求解三体问题 https://www.cnblogs.com/aininot260/p/10039364.html

使用四阶龙格库塔方法求解三体问题（解十二元一阶常微分方程组）

问题简化为三个质点的质量相同的情况

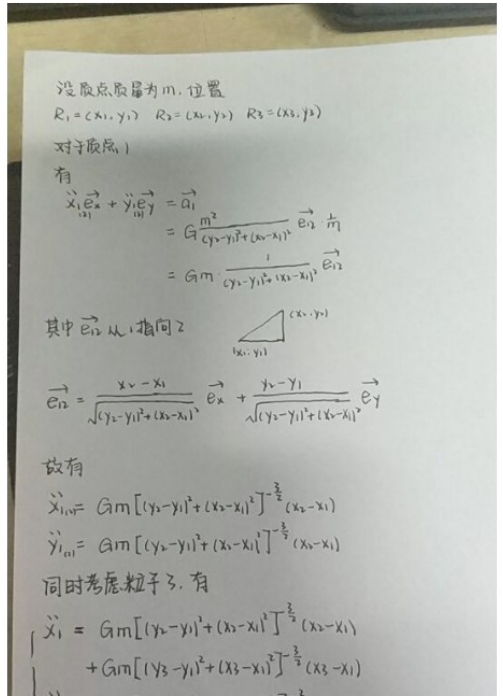
输入所求微分方程组的初值 t0,x1,y1,x2,y2,x3,y3,vx1,vy1,vx2,vy2,vx3,vy3

输入所求微分方程组的微分区间[a,b]

输入所求微分方程组所分解子区间的个数step

输出文件可以直接用Matlab来作图

当时怎么写的我已经完全忘记了，我只记得当时很佩服自己



公告

昵称：静听风吟。
园龄：5年9个月
粉丝：27
关注：2
+加关注

2024年3月

日	一	二	三	四	五	六
25	26	27	28	29	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

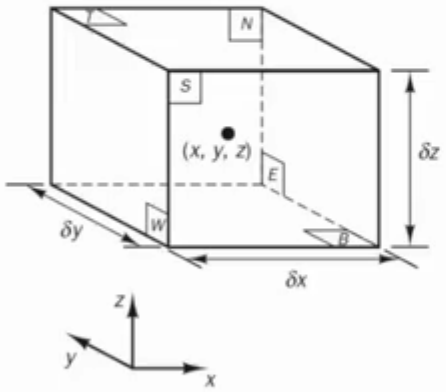
动态规划(28)
分治法(7)
计算几何(2)
课程设计(1)

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

Taichi Lang

Navier-Stokes 方程的构建



(An introduction to Computational Fluid Dynamics, p 10, Fig 2.1)

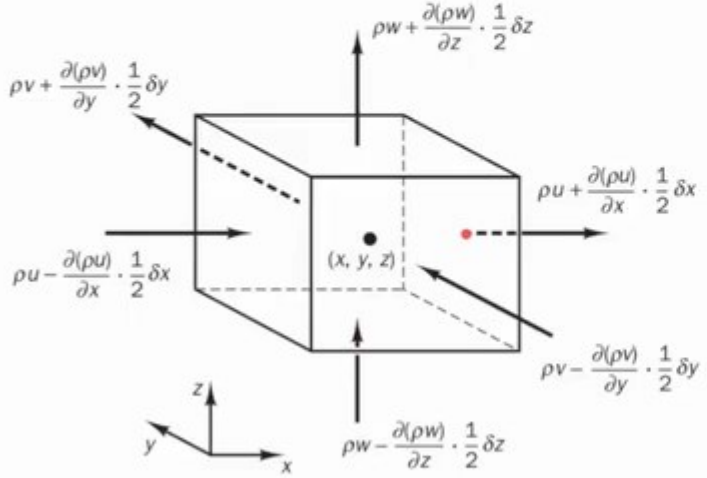
固定微小体积 ➡ 守恒 微分形式
移动微小体积 ➡ 非守恒 微分形式
固定有限体积 ➡ 守恒 积分形式
移动有限体积 ➡ 非守恒 积分形式

- 导出NS方程有多种方法，导出形式各不相同，数学上是可以相互转化的（未必等价）
- 控制容积是微小的流体区域，流体的物理特性定义在容积中心，而各个面上的物理量可以用1 阶泰勒展开
- 连续性假设依然成立，不需要考虑分子原子带来的不连续问题

Taichi Lang

Navier-Stokes 方程

质量守恒：流体在不可压缩的情况下必须进出相等



(An introduction to Computational Fluid Dynamics, p 11, Fig 2.2)

$$\nabla \cdot \vec{u} = 0$$

Taichi Lang

Navier-Stokes 方程

动量守恒：流体动量的变化等于施加在流体上的外力（压力，粘性力）总和

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u}$$

非守恒形

$$\frac{\partial(\rho\vec{u})}{\partial t} + \nabla \cdot (\rho\vec{u}\vec{u}) = -\nabla p + \nabla \cdot (\mu \nabla \vec{u})$$

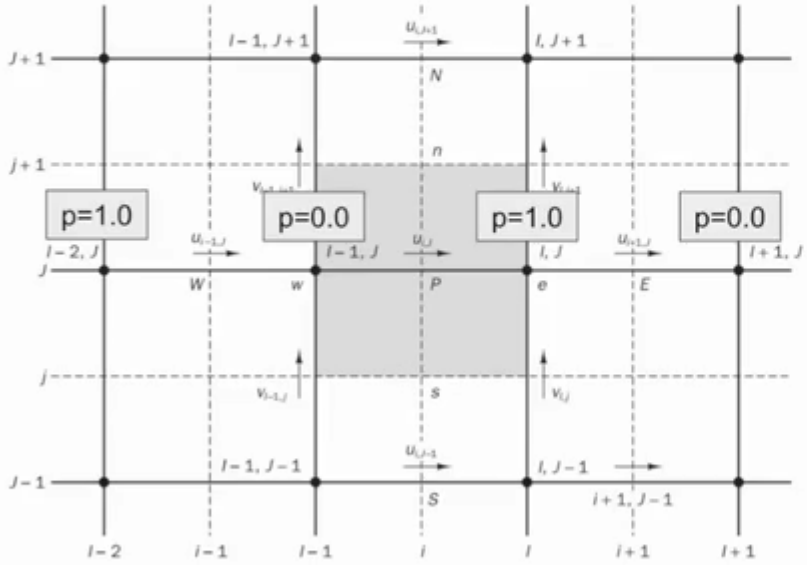
动量变化　　压力　　粘性力

x方向：$$\frac{\partial(\rho u)}{\partial t} + \nabla \cdot (\rho u \vec{u}) = -\frac{\partial p}{\partial x} + \mu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2})$$

y方向：$$\frac{\partial(\rho v)}{\partial t} + \nabla \cdot (\rho v \vec{u}) = -\frac{\partial p}{\partial y} + \mu(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2})$$

Taichi Lang

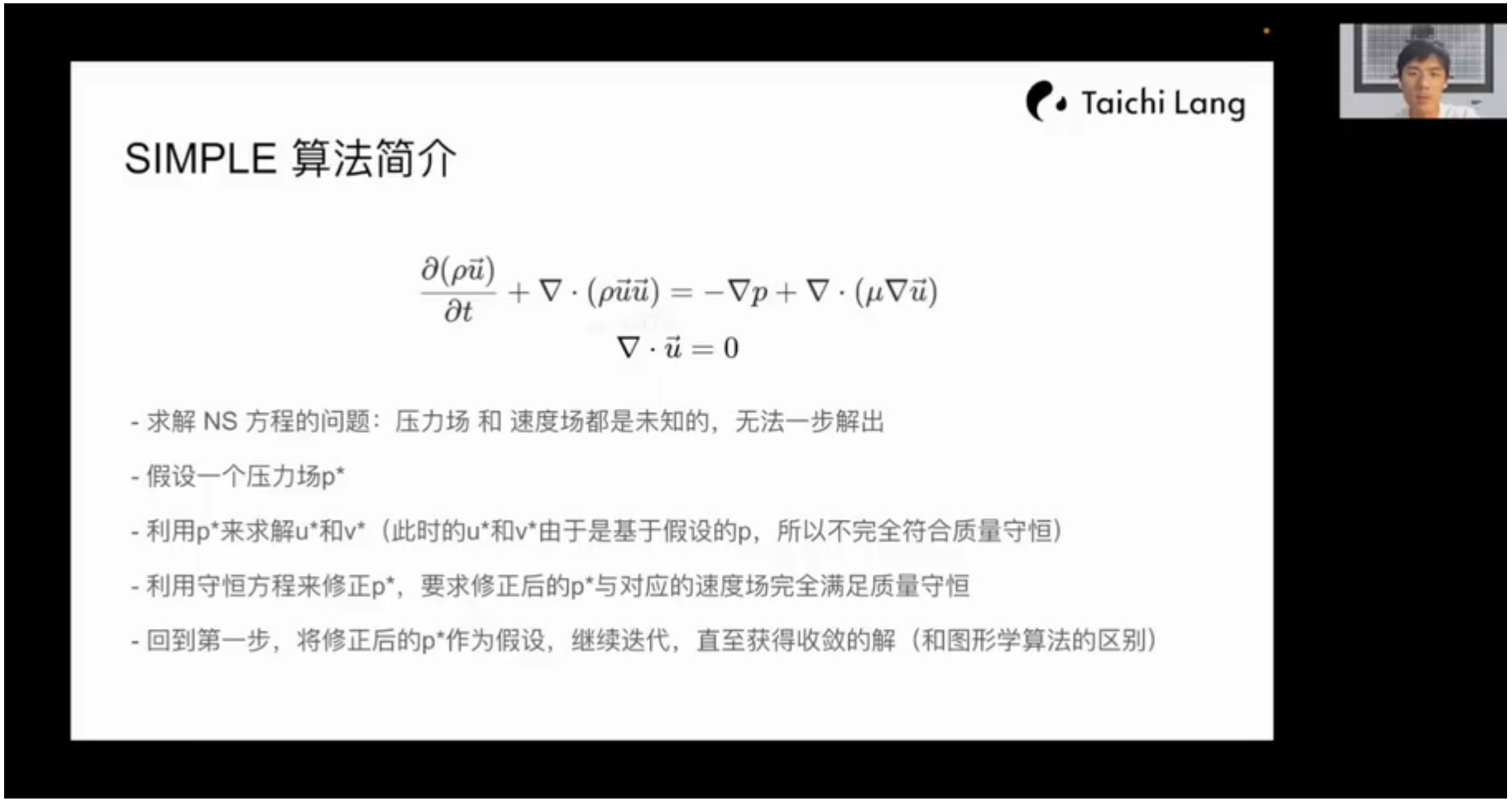
Navier-Stokes 方程的离散化



(An introduction to Computational Fluid Dynamics, p 184, Fig 6.3)

交叉网格（Staggered Grid or MAC Grid）

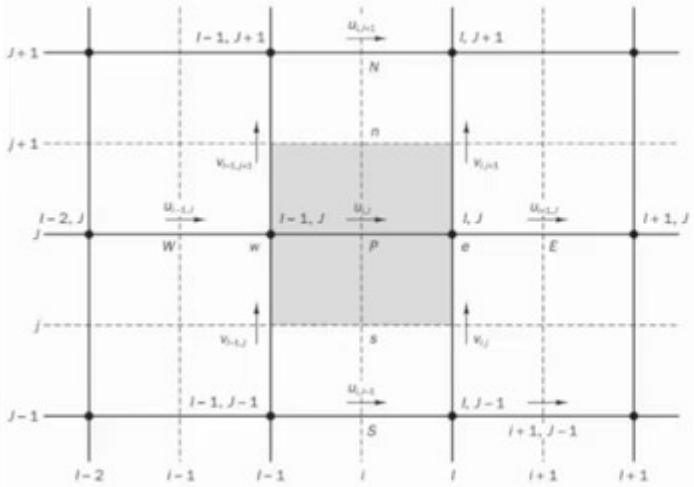
- 标量矢量分开存储
- 解决阶梯状压力波


Taichi Lang
SIMPLE 算法简介
$$\frac{\partial(\rho\vec{u})}{\partial t} + \nabla\cdot(\rho\vec{u}\vec{u}) = -\nabla p + \nabla\cdot(\mu\nabla\vec{u})$$
$$\nabla\cdot\vec{u} = 0$$
- 求解 NS 方程的问题：压力场 和 速度场都是未知的，无法一步解出
- 假设一个压力场p*
- 利用p*来求解u*和v*（此时的u*和v*由于是基于假设的p，所以不完全符合质量守恒）
- 利用守恒方程来修正p*，要求修正后的p*与对应的速度场完全满足质量守恒
- 回到第一步，将修正后的p*作为假设，继续迭代，直至获得收敛的解（和图形学算法的区别）


Taichi Lang
流体数值计算技术
计算流体力学 CFD
Ansys Fluent，OpenFOAM
所广泛采用的数值流体算法
SIMPLE，
SIMPLER，PISO
SPH, DEM
拉格朗日视角：
便于追踪物理量的
流动（Advection）
利于并行计算
守恒特性和不可压
缩性比较难保证
欧拉视角：
便于评价物理量
的空间微分
守恒特性较好
容易丢失流体细
节，需要大量网
格
SPH，WCSPH
MAC，Stable fluid
MPM, PIC, FLIP
Blender，Houdini 等特效软
件常用的模拟算法
计算机图形学 CG

Taichi Lang

动量方程的离散



(An introduction to Computational Fluid Dynamics, p 184, Fig 6.3)

x方向: $\frac{\partial(\rho u)}{\partial t} + \nabla \cdot (\rho u \vec{u}) = -\frac{\partial p}{\partial x} + \mu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2})$

y方向: $\frac{\partial(\rho v)}{\partial t} + \nabla \cdot (\rho v \vec{u}) = -\frac{\partial p}{\partial y} + \mu(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2})$

求解策略:

- CG： Operator Splitting (Advection -> Projection)
- CFD : Prediction -> Correction
- SIMPLE： 先预设一个 p*，然后进行修正；
- 压力的梯度很容易表示，接下来看看其他项

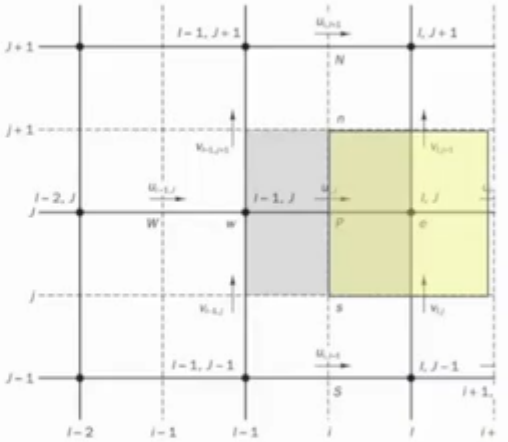
Taichi Lang

压力修正方程的导出

$$a_P u_P = a_E u_E + a_W u_W + a_S u_S + a_N u_N + (p_{I-1,J} - p_{I,J})\frac{A}{dx}$$

$$a_P u^*_P = a_E u^*_E + a_W u^*_W + a_S u^*_S + a_N u^*_N + (p^*_{I-1,J} - p^*_{I,J})\frac{A}{dx}$$

$$a_P u'_P = \boxed{a_E u'_E + a_W u'_W + a_S u'_S + a_N u'_N} + (p'_{I-1,J} - p'_{I,J})\frac{A}{dx}$$



速度修正公式：$u_P = u^*_P + u'_P, u'_P = \frac{1}{a_P}\frac{A}{dx}(p'_{I-1,J} - p'_{I,J})$

连续性条件：$u_{i+1,J} - u_{i,J} + v_{I,j+1} - v_{I,j} = 0$

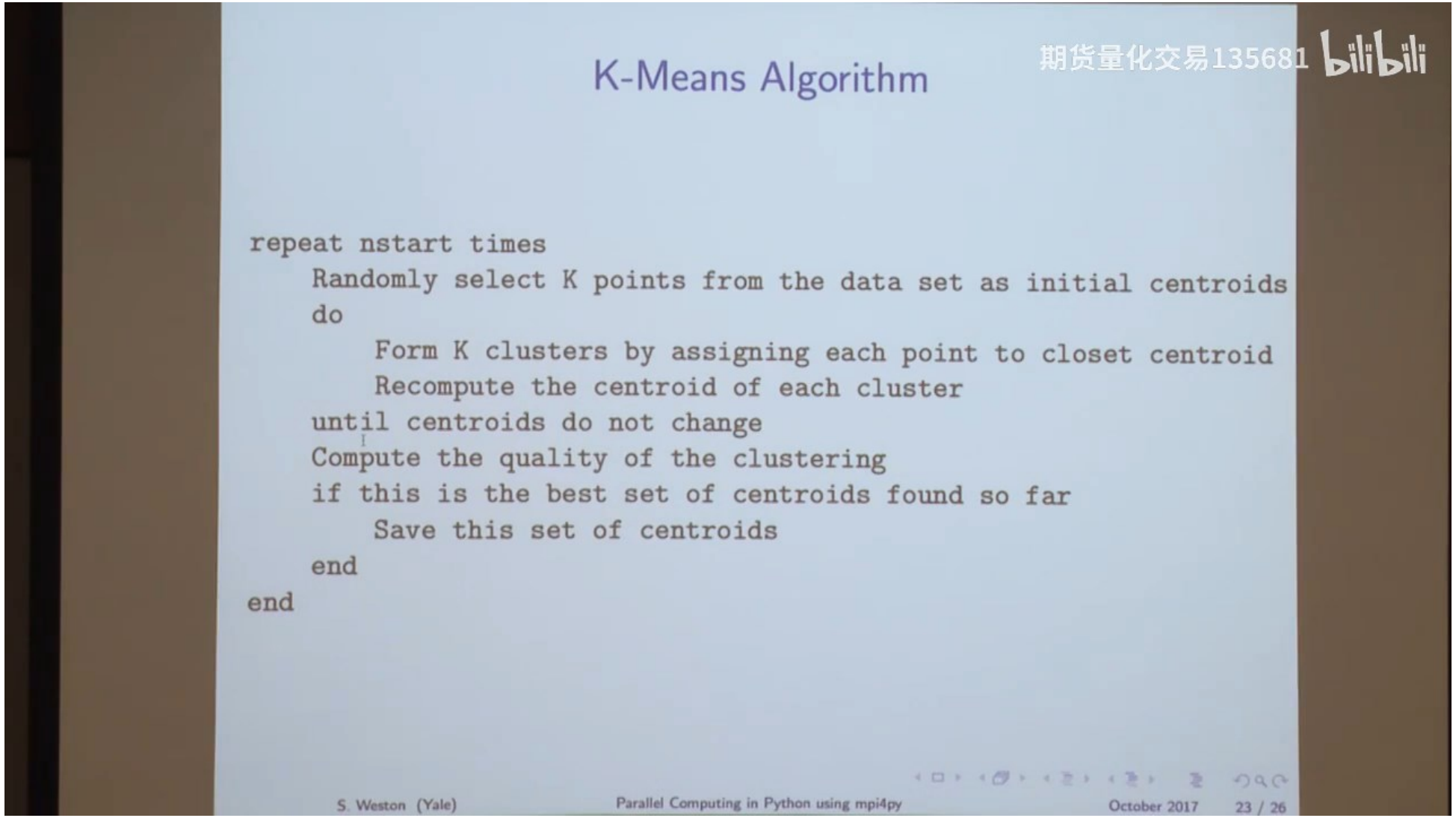
压力修正方程：

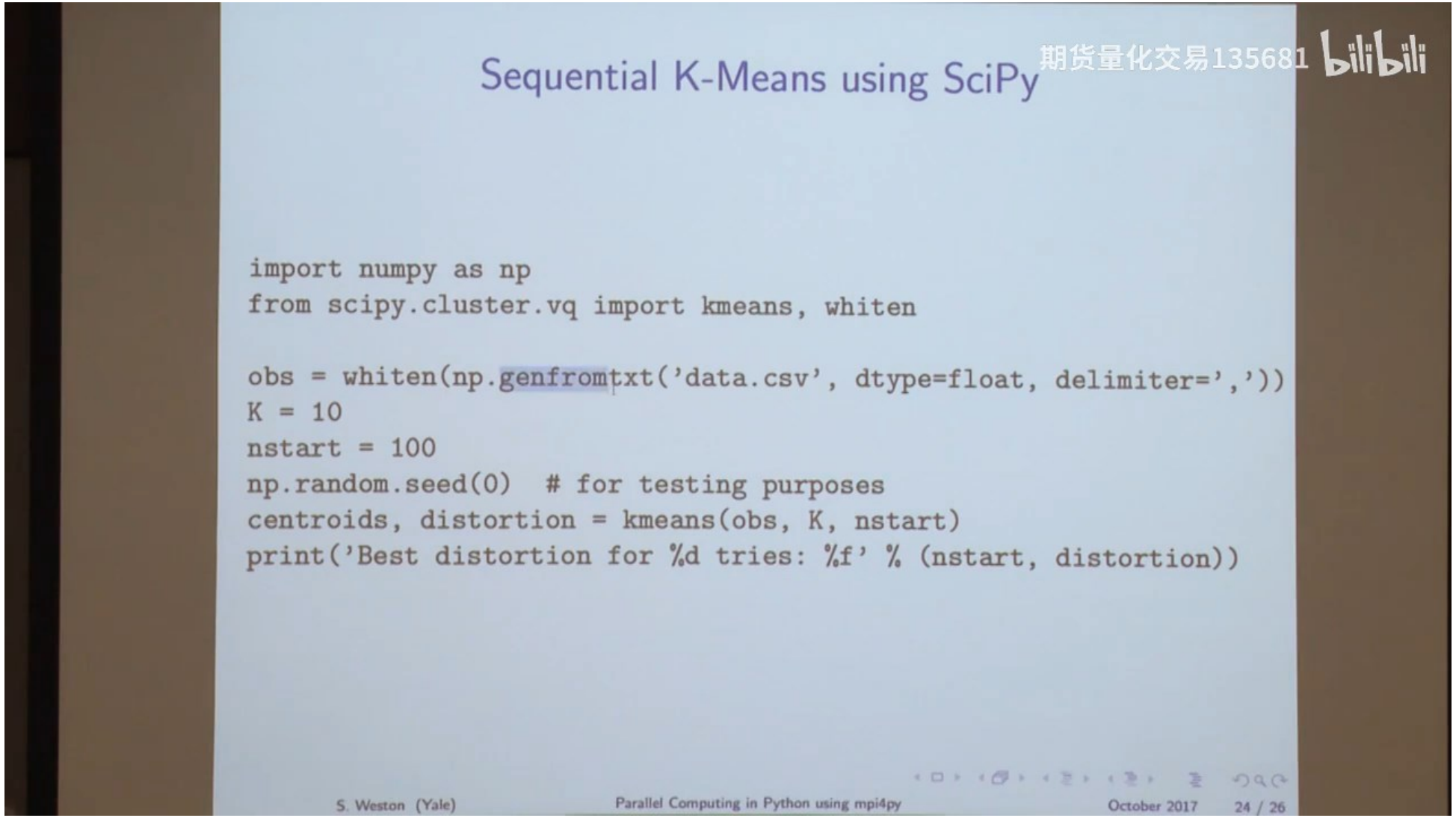
$$\frac{A}{a_{i+1,J}dx}(p'_{I,J} - p'_{I+1,J}) - \frac{A}{a_{i-1,J}dx}(p'_{I-1,J} - p'_{I,J}) + \frac{A}{a_{I,j+1}dy}(p'_{I,J} - p'_{I,J+1}) - \frac{A}{a_{I,j}dy}(p'_{I,J-1} - p'_{I,J}) = -(u^*_{i+1,J} - u^*_{i,J} + v^*_{I,j+1} - v^*_{I,j})$$

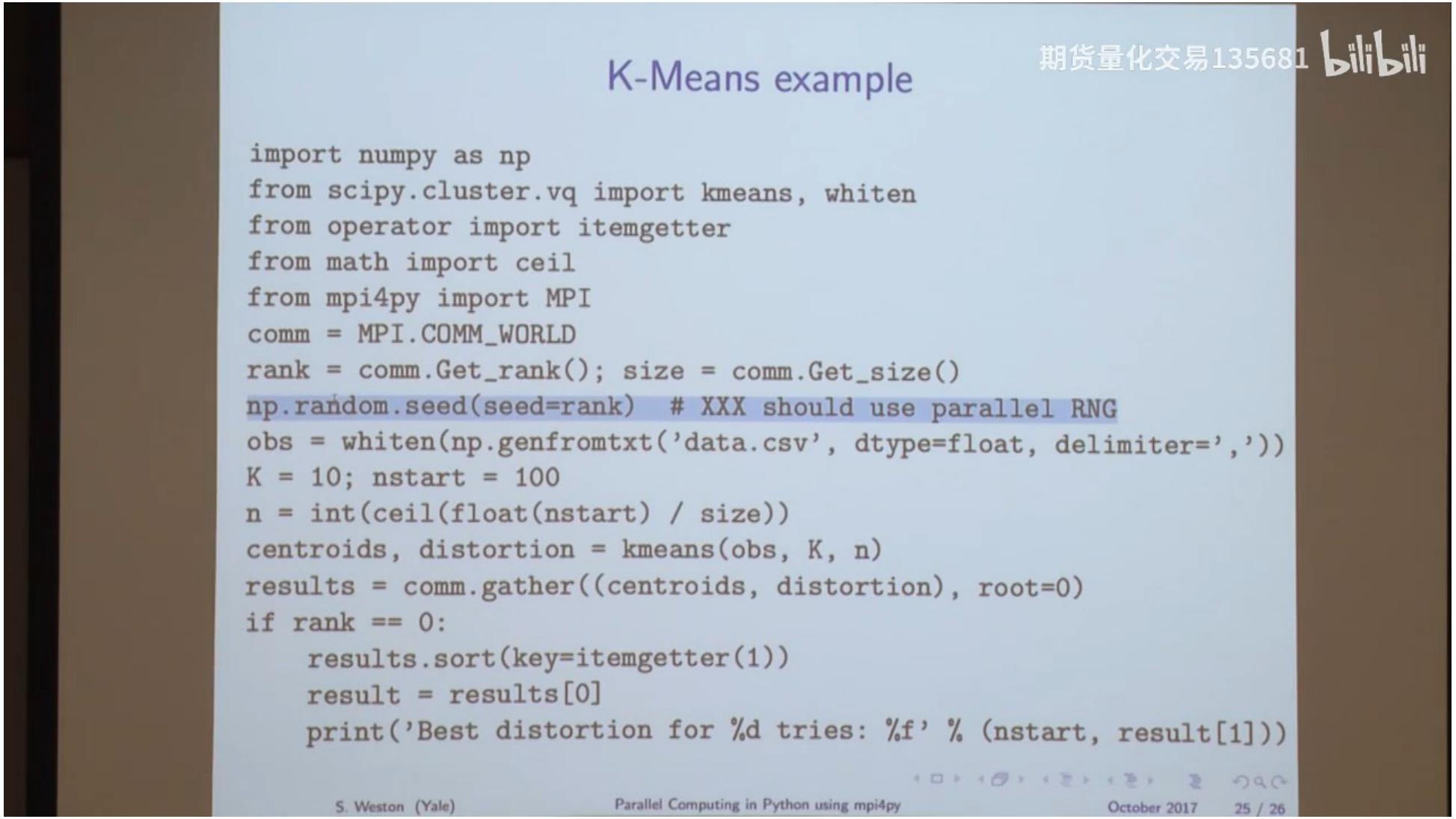
10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

K-means







K-Means example: alternate ending

Instead of sending all of the results to rank 0, we can perform an "allreduce" on the distortion values so that all of the workers know which worker has the best result. Then the winning worker can broadcast its centroids to everyone else.

```
centroids, distortion = kmeans(obs, K, n)
distortion, i = comm.allreduce(distortion, op=MPI.MINLOC)
comm.Bcast([centroids, MPI.FLOAT], root=i)
```

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

Motivation and Introduction

Why is Page Importance Rating important?

- New challenges for information retrieval on the World Wide Web.
  - Huge number of web pages: 150 million by1998
    1000 billion by 2008
  - Diversity of web pages: different topics, different quality, etc.

What is PageRank?

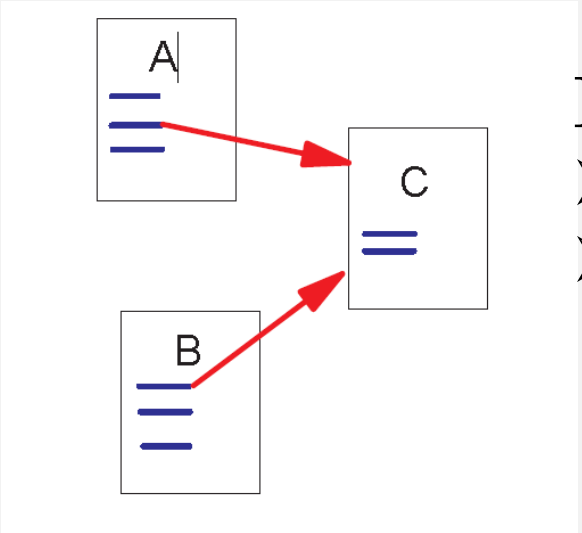
- A method for rating the importance of web pages objectively and mechanically using the link structure of the web.

The History of PageRank

- PageRank was developed by Larry Page (hence the name *Page*-Rank) and Sergey Brin.
- It is first as part of a research project about a new kind of search engine. That project started in 1995 and led to a functional prototype in 1998.
- Shortly after, Page and Brin founded Google.
- 16 billion…

Link Structure of the Web

- 150 million web pages → 1.7 billion links



Backlinks and Forward links:

- A and B are C's backlinks
- C is A and B's forward link

Intuitively, a webpage is important if it has a lot of backlinks.

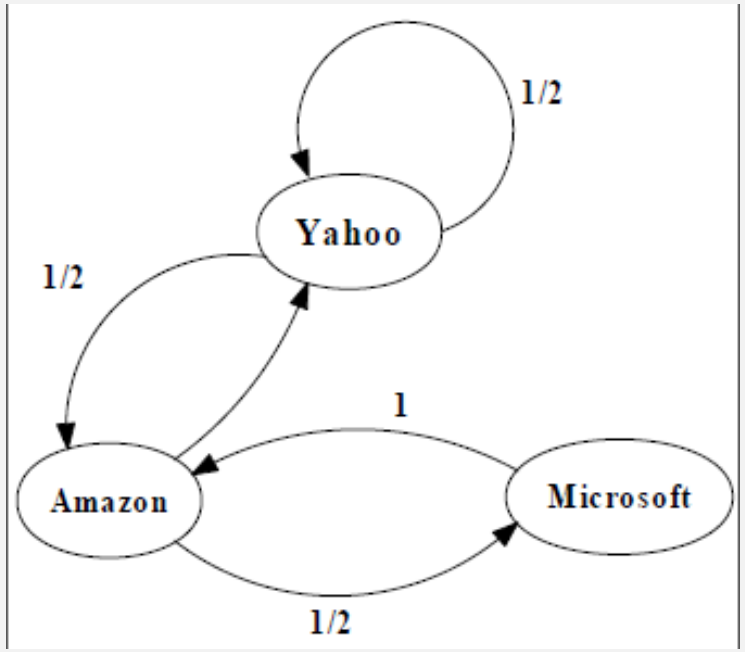
What if a webpage has only one link off www.yahoo.com?

A Simple Version of PageRank

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v}$$

- u: a web page
- B_u: the set of u's backlinks
- N_v: the number of forward links of page v
- c: the normalization factor to make $||R||_{L1} = 1$ ($||R||_{L1} = |R_1 + \ldots + R_n|$)

An example of Simplified PageRank



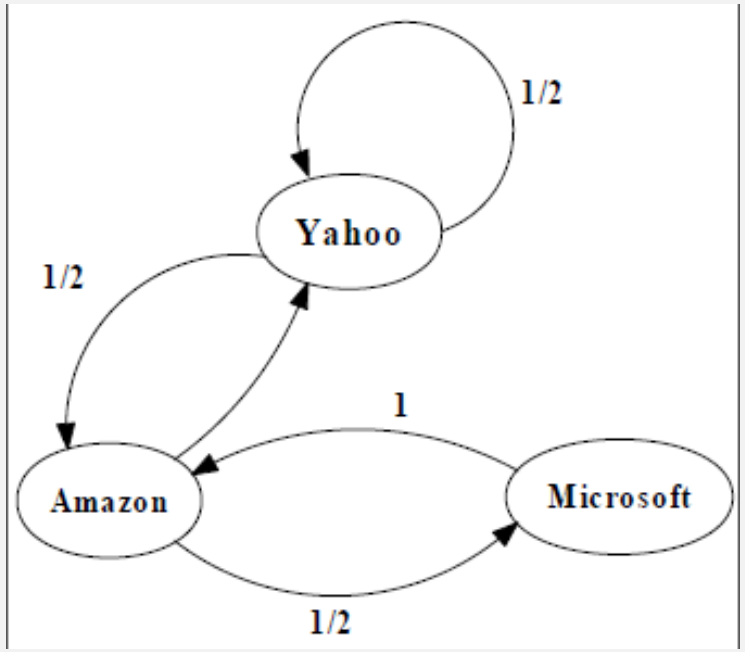
$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

PageRank Calculation: first iteration

An example of Simplified PageRank



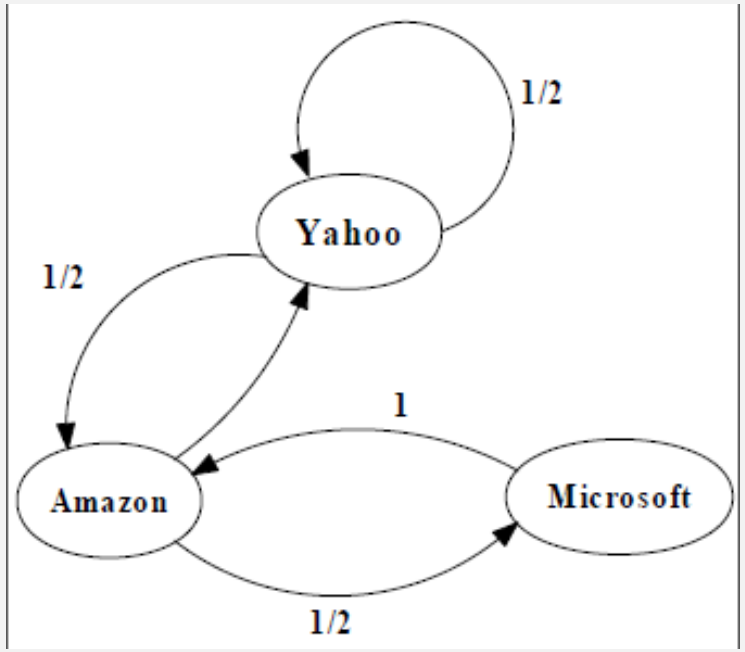
$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 5/12 \\ 1/3 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix}$$

PageRank Calculation: second iteration

An example of Simplified PageRank



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

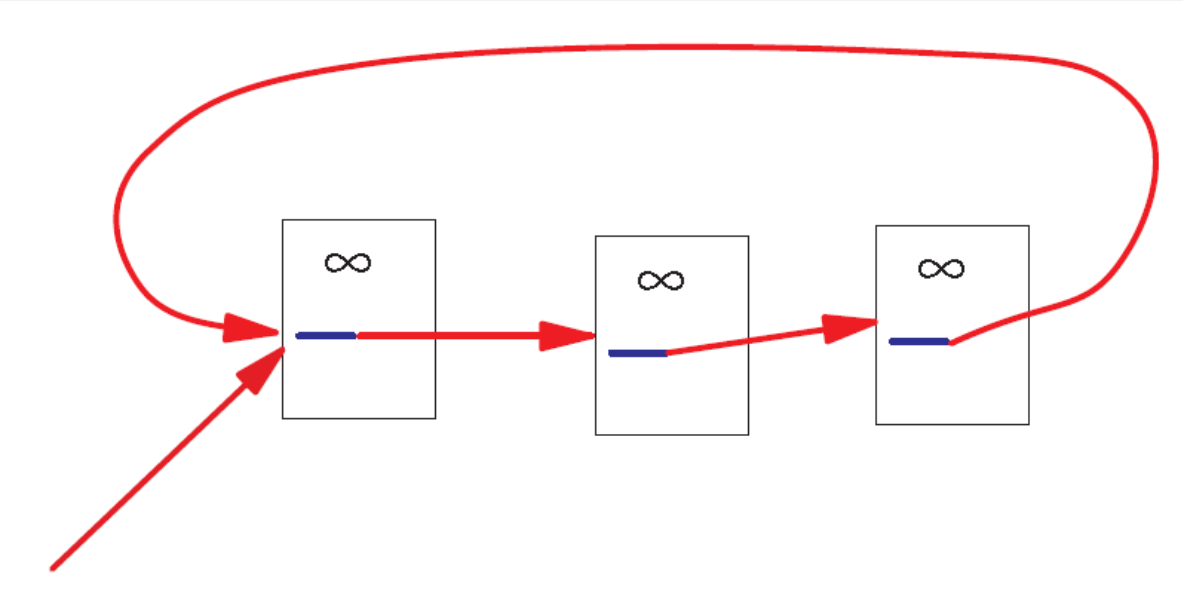
$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 3/8 \\ 11/24 \\ 1/6 \end{bmatrix} \begin{bmatrix} 5/12 \\ 17/48 \\ 11/48 \end{bmatrix} \ldots \begin{bmatrix} 2/5 \\ 2/5 \\ 1/5 \end{bmatrix}$$

Convergence after some iterations

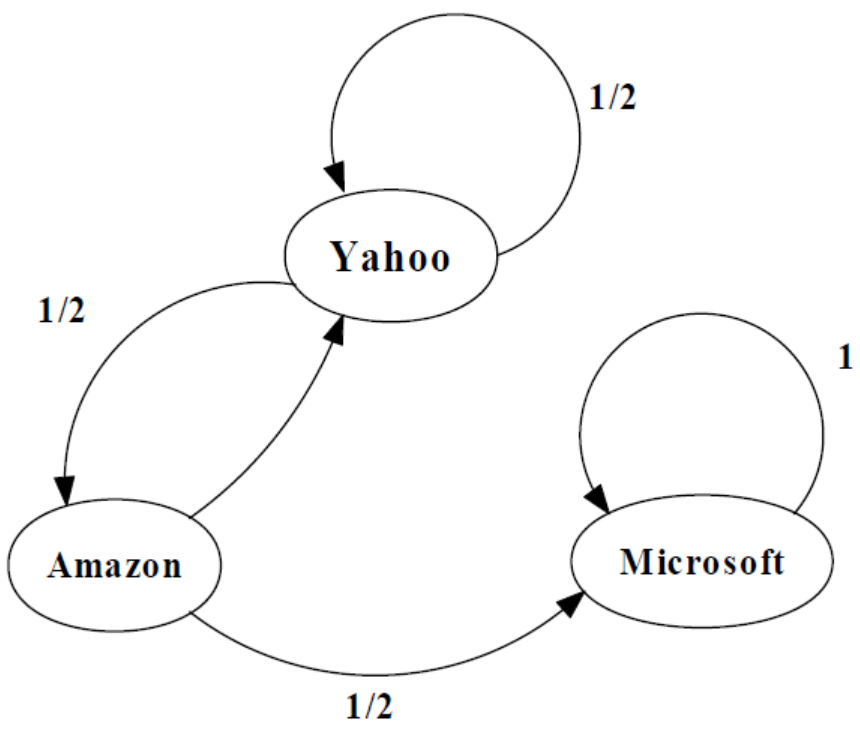
A Problem with Simplified PageRank

A loop:



During each iteration, the loop accumulates rank but never distributes rank to other pages!

An example of the Problem

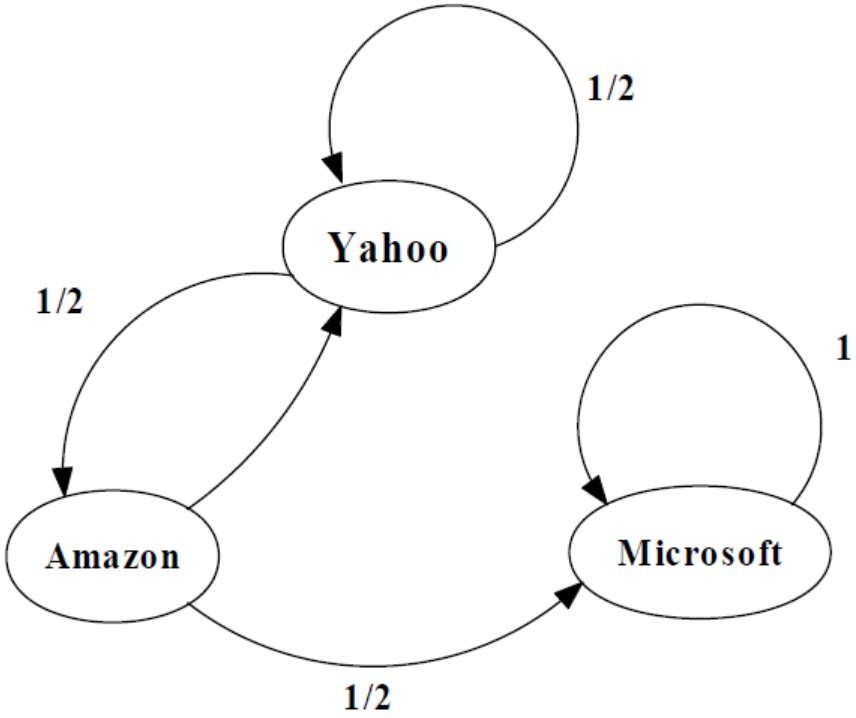


$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 1/3 \\ 1/6 \\ 1/2 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

An example of the Problem

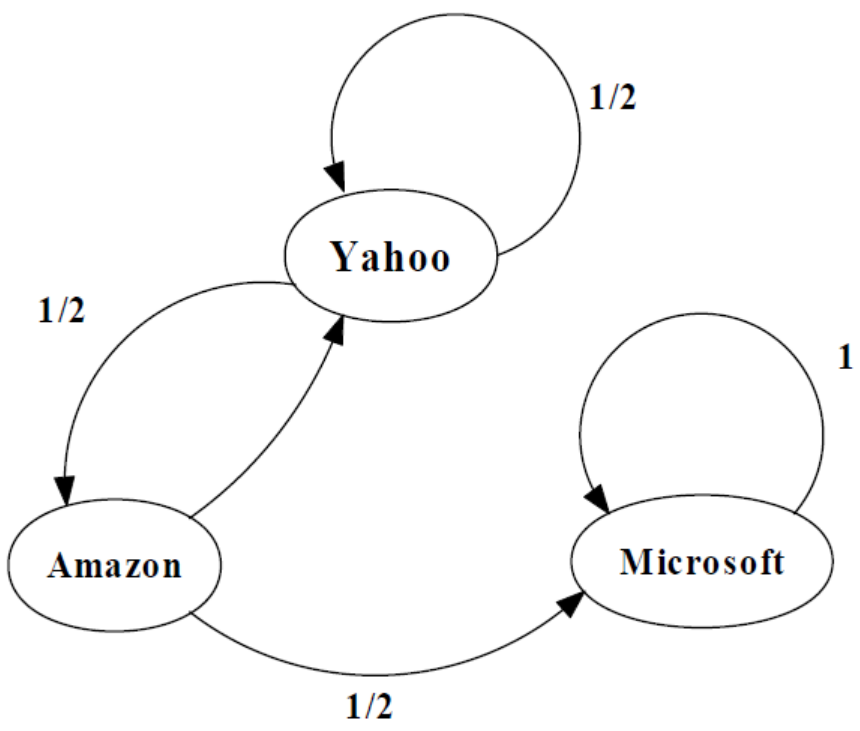


$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 1/4 \\ 1/6 \\ 7/12 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/6 \\ 1/2 \end{bmatrix}$$

An example of the Problem



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 5/24 \\ 1/8 \\ 2/3 \end{bmatrix} \begin{bmatrix} 1/6 \\ 5/48 \\ 35/48 \end{bmatrix} \ldots. \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Random Walks in Graphs

The Random Surfer Model

- The simplified model: the standing probability distribution of a random walk on the graph of the web. simply keeps clicking successive links at random

The Modified Model

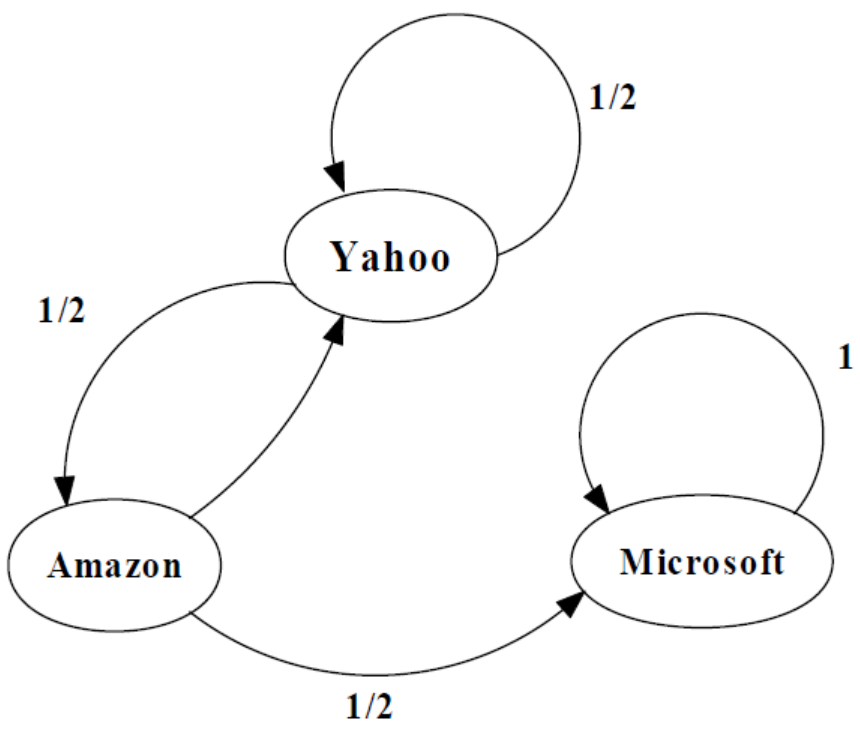
- The modified model: the "random surfer" simply keeps clicking successive links at random, but periodically "gets bored" and jumps to a random page based on the distribution of E

Modified Version of PageRank

$$R'(u) = c_1 \sum_{v \in B_u} \frac{R'(v)}{N_v} + c_2 E(u)$$

E(u): a distribution of ranks of web pages that "users" jump to when they "gets bored" after successive links at random.

An example of Modified PageRank



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$C_1 = 0.8 \qquad C_2 = 0.2$

$$\begin{bmatrix} 0.333 \\ 0.333 \\ 0.333 \end{bmatrix} \begin{bmatrix} 0.333 \\ 0.200 \\ 0.467 \end{bmatrix} \begin{bmatrix} 0.280 \\ 0.200 \\ 0.520 \end{bmatrix} \begin{bmatrix} 0.259 \\ 0.179 \\ 0.563 \end{bmatrix} \ldots \begin{bmatrix} 7/33 \\ 5/33 \\ 21/33 \end{bmatrix}$$

Distributed-ML-PySpark-master

新加卷 (D:) › myCodes › HPCprojects › SourceCodes › pyspark › Distributed-ML-PySpark-master › graph_computation

名称	修改日期	类型	大小
pagerank.py	2023,3/18,周六 19:33	PY 文件	3 KB
transitive_closure.py	2023,3/18,周六 19:33	PY 文件	2 KB

```python
def computeContribs(neighbors: ResultIterable[int], rank
    # Calculates the contribution(rank/num_neighbors) of each vertex, and send it to its neighbours.
    num_neighbors = len(neighbors)
    for vertex in neighbors:
        yield (vertex, rank / num_neighbors)

if __name__ == "__main__":
    # Initialize the spark context.
    spark = SparkSession\
        .builder\
        .appName("PageRank")\
        .master("local[%d]" % n_threads)\
        .getOrCreate()

    # link: (source_id, dest_id)
    links = spark.sparkContext.parallelize(
        [(1, 2), (1, 3), (2, 3), (3, 1)]
    )

    # drop duplicate links and convert to
    adj_list = links.distinct().groupByKey

    # count the number of vertexes
    n_vertexes = adj_list.count()

    # init the rank of each vertex, the
    ranks = adj_list.map(lambda vertex_r

    # Calculates and updates vertex rank
    for t in range(n_iterations):
        # Calculates the contribution(ra
        contribs = adj_list.join(ranks).
            vertex_neighbors_rank[1][0],
        ))

        # Re-calculates rank of each ver
        ranks = contribs.reduceByKey(add

    # Collects all ranks of vertexs and dump them to console.
    for (vertex, rank) in ranks.collect():
        print("%s has rank: %s." % (vertex, rank))

    spark.stop()


# 1 has rank: 0.38891305880091237.
# 2 has rank: 0.214416470596171.
# 3 has rank: 0.3966704706029163.
```

C:\Windows\System32\cmd.exe

```
Microsoft Windows [版本 10.0.19045.2728]
(c) Microsoft Corporation。保留所有权利。

E:\myJupyter\Distributed-ML-PySpark-master\graph_computation>python pagerank.py
23/03/29 15:38:25 WARN Shell: Did not find winutils.exe: java.io.FileNotFoundException: java.io.FileNotFoundException: H
ADOOP_HOME and hadoop.home.dir are unset. -see https://wiki.apache.org/hadoop/WindowsProblems
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/03/29 15:38:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
23/03/29 15:38:29 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes
1 has rank: 0.38891305880091237.
2 has rank: 0.214416470596171.
3 has rank: 0.3966704706029163.

E:\myJupyter\Distributed-ML-PySpark-master\graph_computation>成功: 已终止 PID 9396 (属于 PID 15996 子进程)的进程。
成功: 已终止 PID 15996 (属于 PID 10212 子进程)的进程。
成功: 已终止 PID 10212 (属于 PID 7256 子进程)的进程。
```

Dangling Links

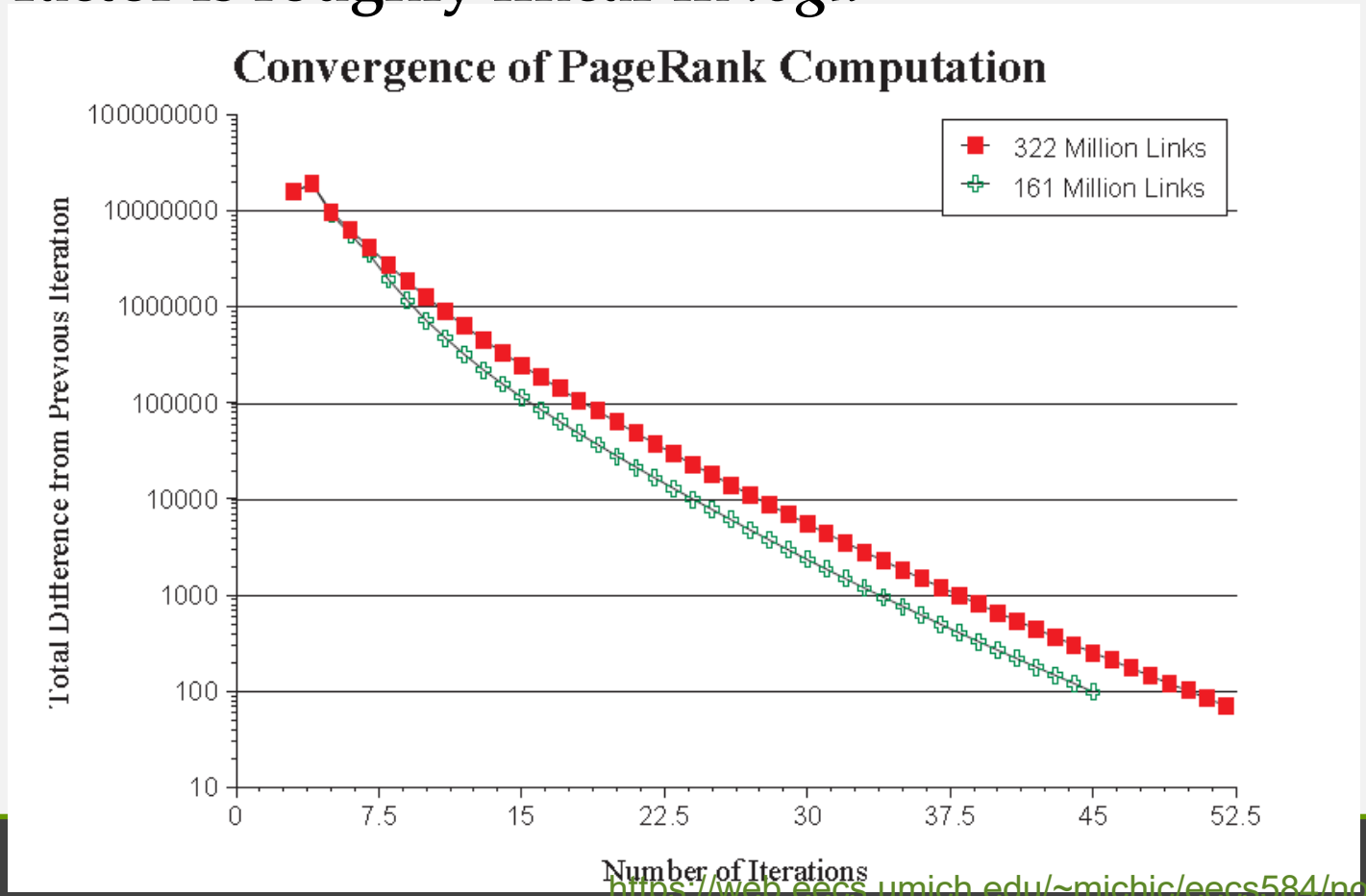
- Links that point to any page with no outgoing links
- Most are pages that have not been downloaded yet
- Affect the model since it is not clear where their weight should be distributed
- Do not affect the ranking of any other page directly
- Can be simply removed before pagerank calculation and added back afterwards

PageRank Implementation

- **Convert each URL into a unique integer and store each hyperlink in a database using the integer IDs to identify pages**
- **Sort the link structure by ID**
- **Remove all the dangling links from the database**
- **Make an initial assignment of ranks and start iteration**
  - Choosing a good initial assignment can speed up the pagerank
- **Adding the dangling links back.**

Convergence Property

- PR (322 Million Links): 52 iterations
- PR (161 Million Links): 45 iterations
- Scaling factor is roughly linear in *logn*



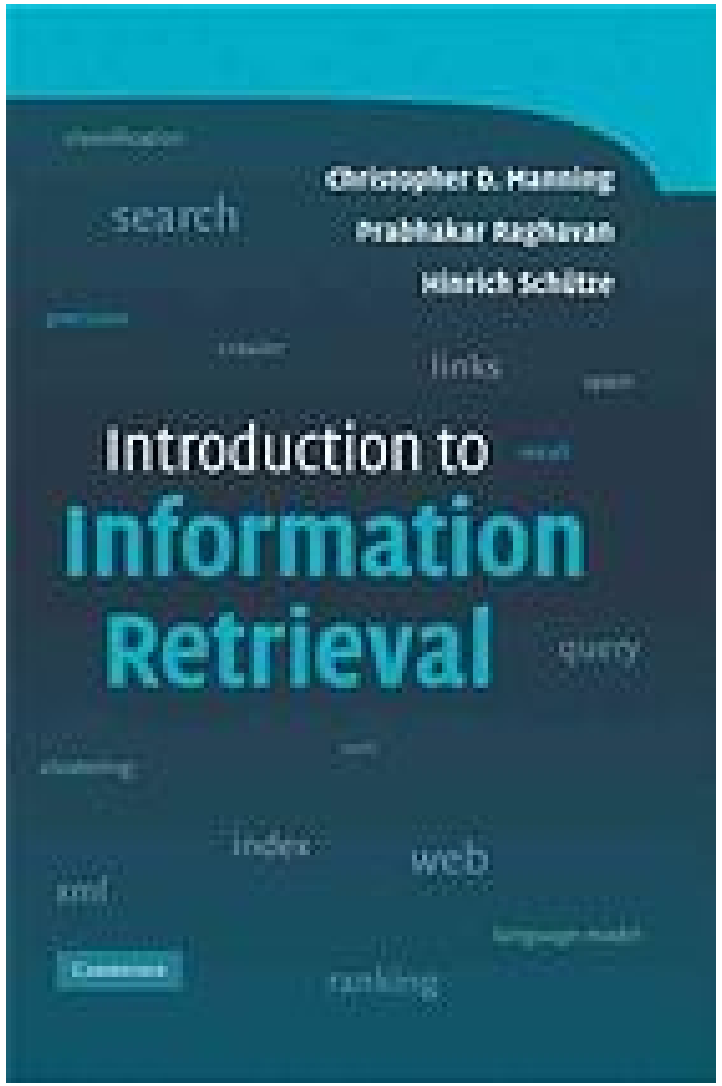
https://web.eecs.umich.edu/~michjc/eecs584/notes/lecture19-pagerank.ppt

Convergence Property

The Web is an expander-like graph

- Theory of random walk: a random walk on a graph is said to be rapidly-mixing if it quickly converges to a limiting distribution on the set of nodes in the graph. A random walk is rapidly-mixing on a graph if and only if the graph is an expander graph.

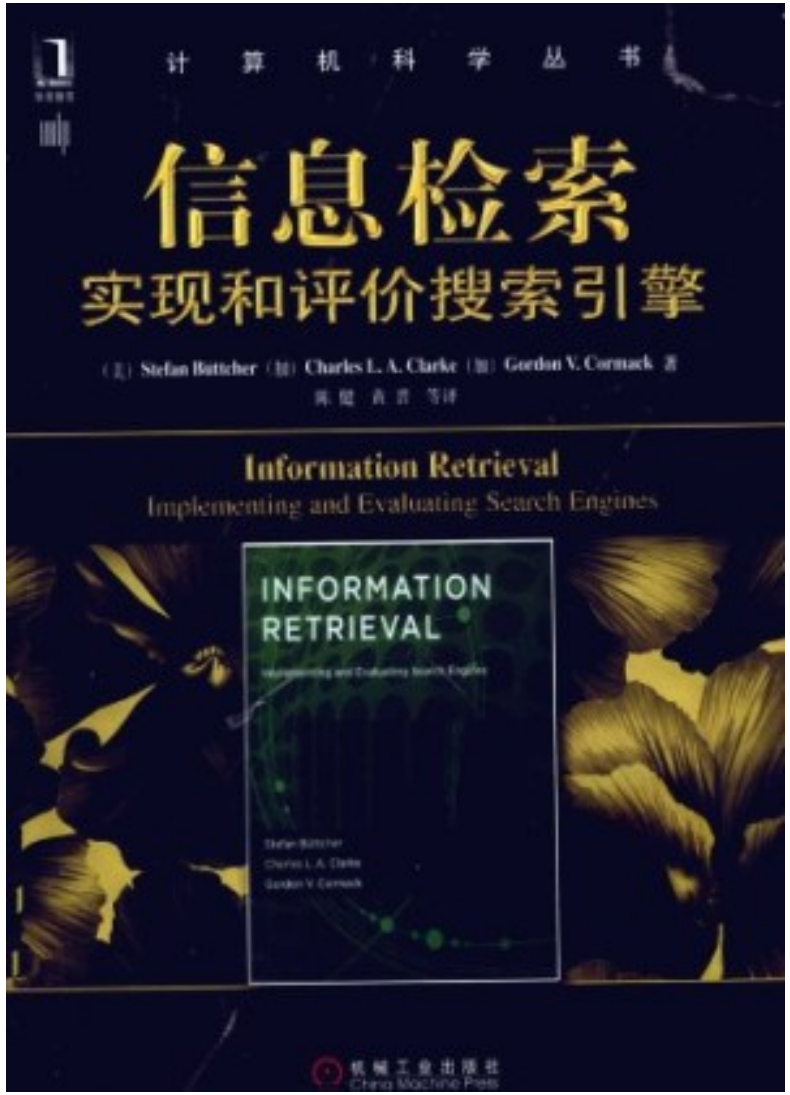
- Expander graph: every subset of nodes S has a neighborhood (set of vertices accessible via outedges emanating from nodes in S) that is larger than some factor α times of |S|. A graph has a good expansion factor if and only if the largest eigenvalue is sufficiently larger than the second-largest eigenvalue.



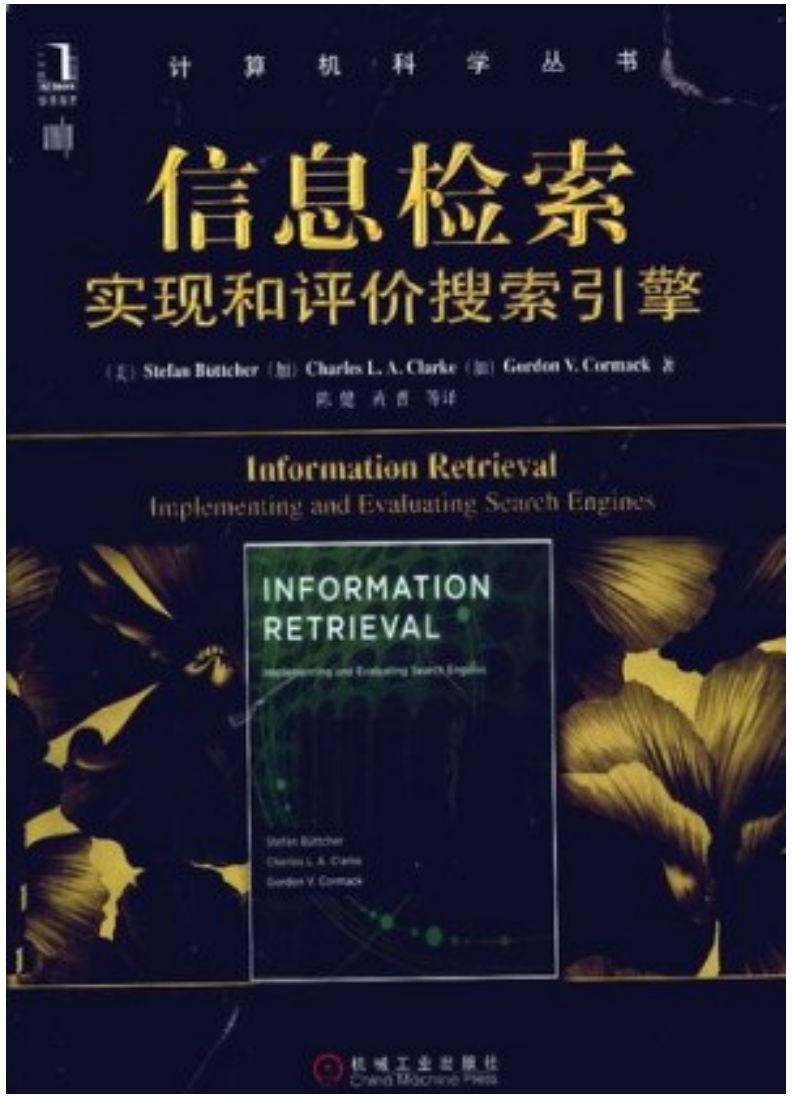
- Introduction to information retrieval
- *Christopher D Manning*, *Prabhakar Raghavan*, *Hinrich Schutze*
- Cambridge University Press
- **2008**

- 信息检索导论
- *Christopher D.Manning*, *Hinrich Schütze*, *Prabhakar Raghavan*
- **2010**
- 人民邮电出版社

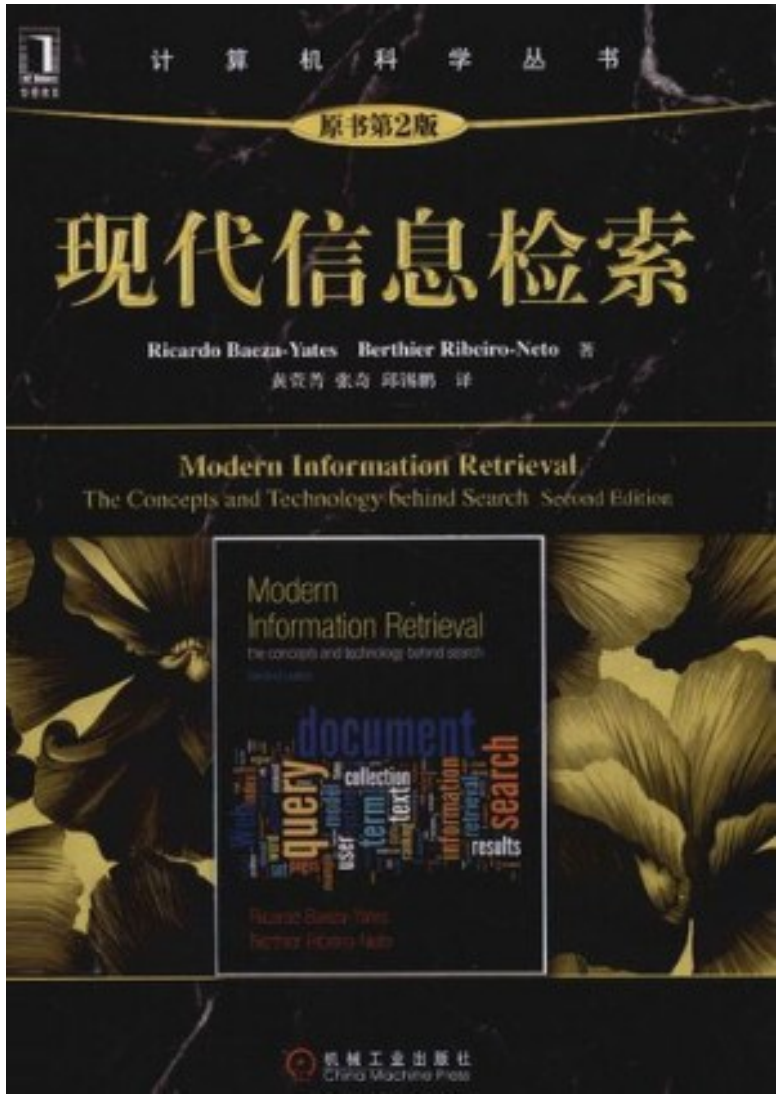
- 信息检索导论（修订版）
- *普拉巴卡尔•拉格万（Prabhakar Raghavan）/ 欣里希•舒策（Hinrich Schütze）/ 克里斯托夫•曼宁（Christopher Manning）*
- 人民邮电出版社
- **2019**



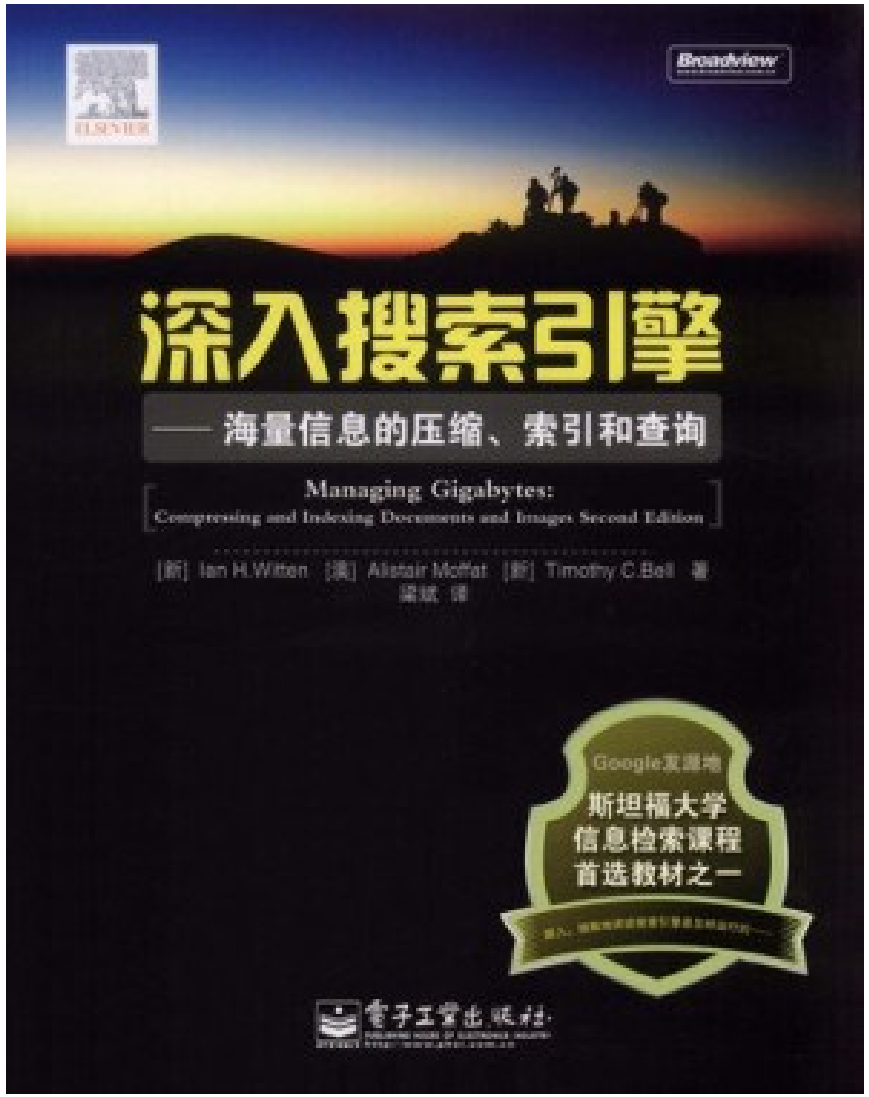
- 信息检索：实现和评价搜索引擎
- *Stefan Büttcher*, *Charles L.A.Clarke*, *Gordon V. Cormack*
- 机械工业出版社
- **2012**



- 搜索引擎与信息检索教程
- *袁津生，等*
- 中国水利水电出版社
- **2008**

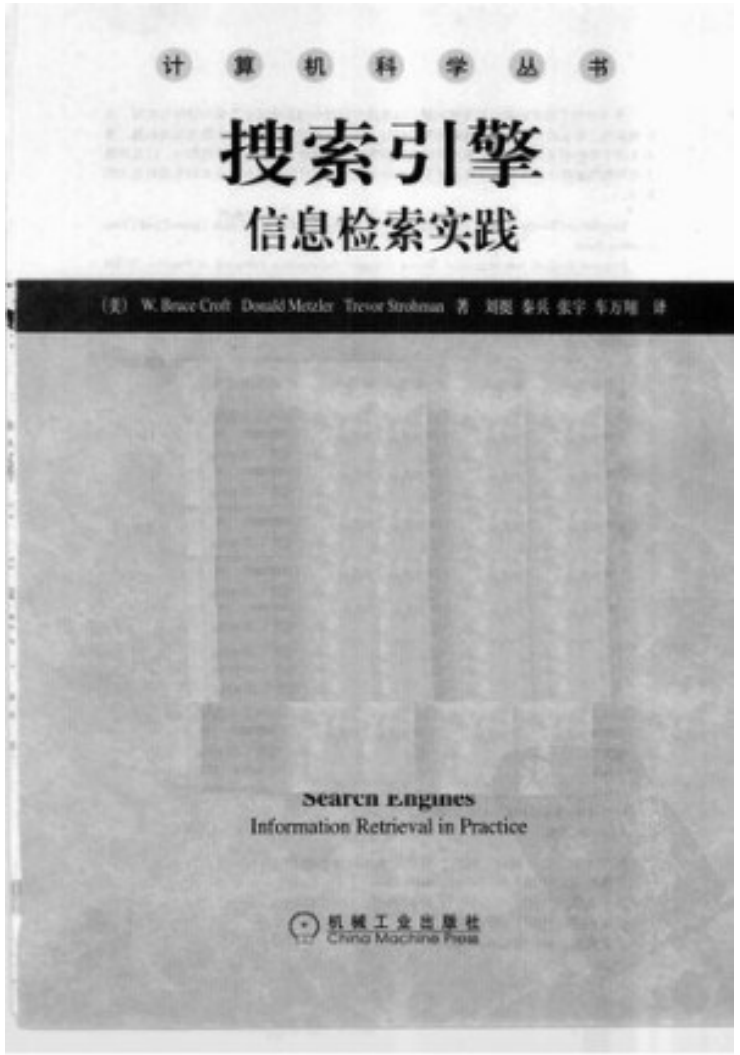


- 现代信息检索
- *Ricardo Baeza-Yates*, *Berthier Ribeiro-Neto*
- 机械工业出版社
- **2012**



- 深入搜索引擎: 海量信息的压缩、索引和查询
- *ian H. Witten*, *Alistair Moffat*, *Timothy C. Bell*
- 电子工业出版社
- **2009**

- 搜索引擎: 原理、技术与系统
- *李晓明*
- 科学出版社
- **2005**

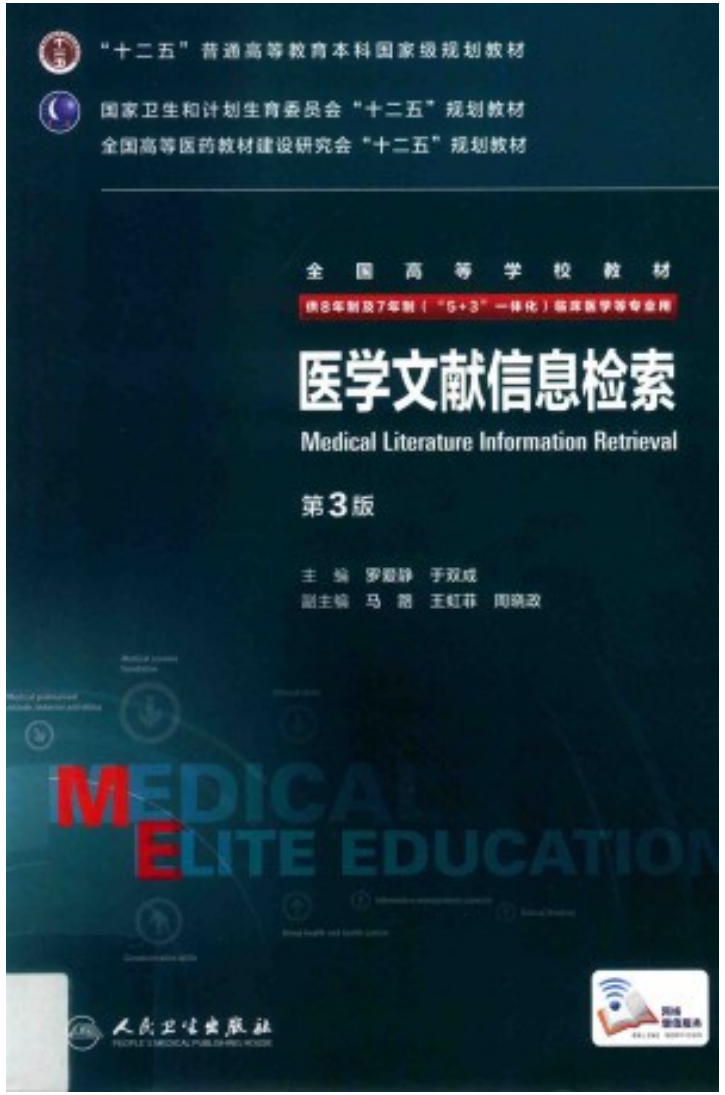


- 搜索引擎: 信息检索实践
- W.Bruce Croft, Donald Metzler, Trevor Strohman
- 机械工业出版社
- **2010**

- 信息检索 (高等院校通识教育“十二五”规划教材)
- *明均仁*
- 人民邮电出版社
- **2014**

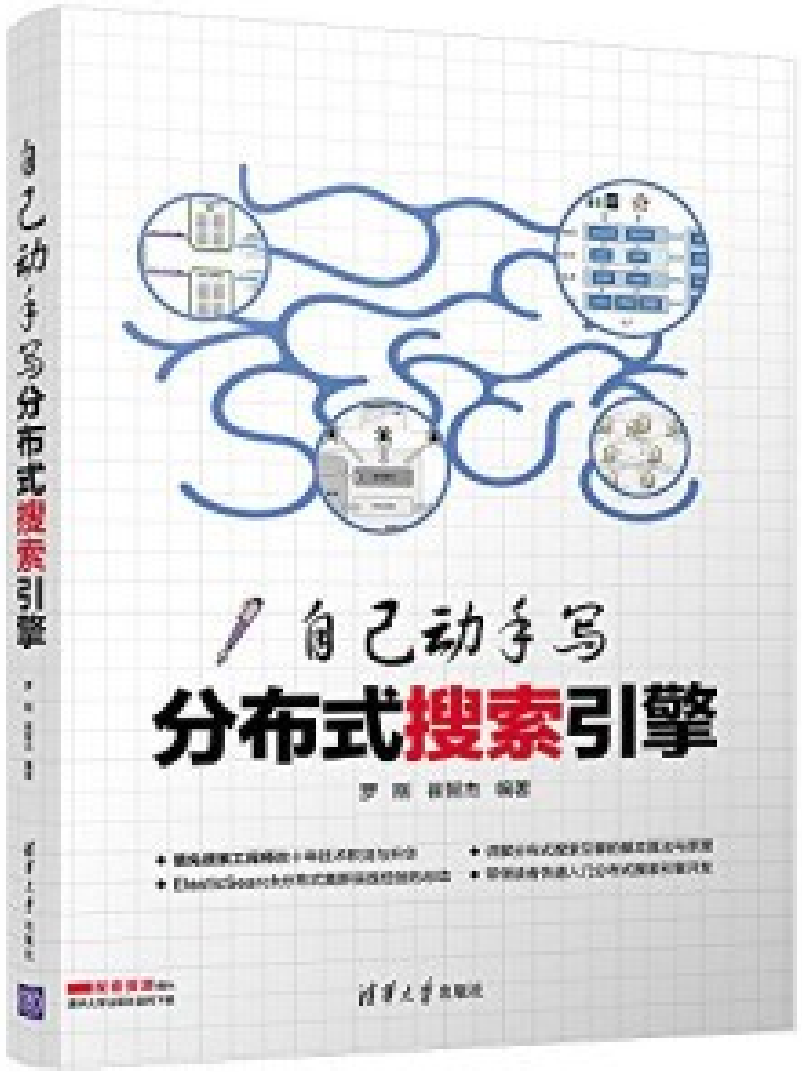
- 人工智能之信息检索与推荐
- *清华大学人工智能研究院，北京智源人工智能研究院，清华-工程院知识智能联合研究中心，Aminers*

- **2019**



- 医学文献信息检索 第3版
- *罗爱静, 于双城*
- 人民卫生出版社
- **2015**

- 图书馆文献信息检索与利用研究
- *梅梅*
- 北京工业大学出版社
- **2021**



- 自己动手写分布式搜索引擎
- *罗刚*
- 清华大学出版社
- **2017**

- 自制搜索引擎
- *山田浩之*
- 人民邮电出版社
- **2016**

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

梯度下降法，或最速梯度下降法？？

附录 A　梯度下降法

梯度下降法（gradient descent）或最速下降法（steepest descent）是求解无约束最优化问题的一种最常用的方法，具有实现简单的优点。梯度下降法是迭代算法，每一步需要求解目标函数的梯度向量。

假设 $f(x)$ 是 $\mathbf{R}^n$ 上具有一阶连续偏导数的函数。要求解的无约束最优化问题是

$$\min_{x \in \mathbf{R}^n} f(x) \tag{A.1}$$

x^* 表示目标函数 $f(x)$ 的极小点。

梯度下降法是一种迭代算法。选取适当的初值 $x^{(0)}$，不断迭代，更新 x 的值，进行目标函数的极小化，直到收敛。由于负梯度方向是使函数值下降最快的方向，在迭代的每一步，以负梯度方向更新 x 的值，从而达到减少函数值的目的。

由于 $f(x)$ 具有一阶连续偏导数，若第 k 次迭代值为 $x^{(k)}$，则可将 $f(x)$ 在 $x^{(k)}$ 附近进行一阶泰勒展开：

$$f(x) = f(x^{(k)}) + g_k^{\mathrm{T}}(x - x^{(k)}) \tag{A.2}$$

这里，$g_k = g(x^{(k)}) = \nabla f(x^{(k)})$ 为 $f(x)$ 在 $x^{(k)}$ 的梯度。

求出第 $k+1$ 次迭代值 $x^{(k+1)}$：

$$x^{(k+1)} \leftarrow x^{(k)} + \lambda_k p_k \tag{A.3}$$

其中，p_k 是搜索方向，取负梯度方向 $p_k = -\nabla f(x^{(k)})$，λ_k 是步长，由一维搜索确定，即 λ_k 使得

$$f(x^{(k)} + \lambda_k p_k) = \min_{\lambda \geqslant 0} f(x^{(k)} + \lambda p_k) \tag{A.4}$$

算法 A.1（梯度下降法）

输入：目标函数 $f(x)$，梯度函数 $g(x) = \nabla f(x)$，计算精度 ε；

输出：$f(x)$ 的极小点 x^*。

（1）取初始值 $x^{(0)} \in \mathbf{R}^n$，置 $k = 0$。

（2）计算 $f(x^{(k)})$。

（3）计算梯度 $g_k = g(x^{(k)})$，当 $\|g_k\| < \varepsilon$ 时，停止迭代，令 $x^* = x^{(k)}$；否则，令 $p_k = -g(x^{(k)})$，求 λ_k，使

$$f(x^{(k)} + \lambda_k p_k) = \min_{\lambda \geqslant 0} f(x^{(k)} + \lambda p_k)$$

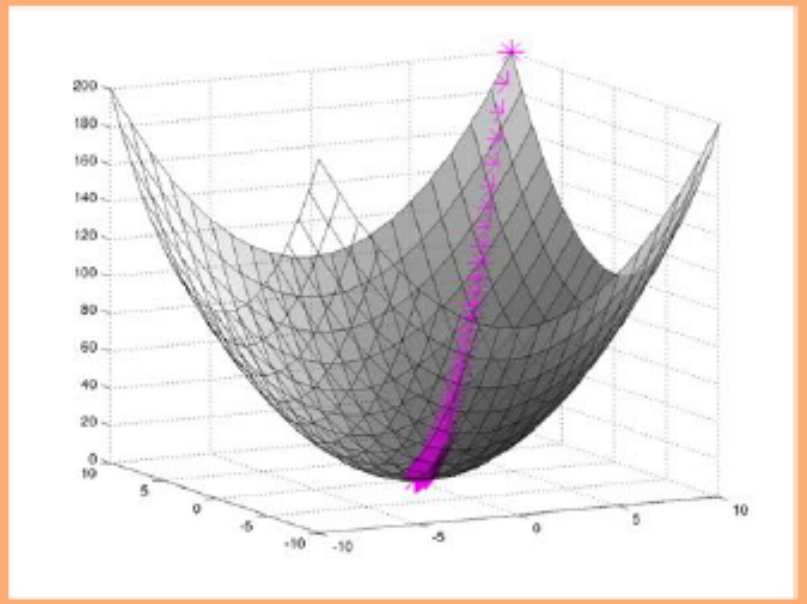
（4）置 $x^{(k+1)} = x^{(k)} + \lambda_k p_k$，计算 $f(x^{(k+1)})$

当 $\|f(x^{(k+1)}) - f(x^{(k)})\| < \varepsilon$ 或 $\|x^{(k+1)} - x^{(k)}\| < \varepsilon$ 时，停止迭代，令 $x^* = x^{(k+1)}$

Gradient Descent

Algorithm 1 Gradient Descent

1: **procedure** GD($\mathcal{D}$, $\boldsymbol{\theta}^{(0)}$)
2: $\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^{(0)}$
3: $\quad$**while** not converged **do**
4: $\quad\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
5: $\quad$**return** $\boldsymbol{\theta}$



附录 B　牛顿法和拟牛顿法

牛顿法（Newton method）和拟牛顿法（quasi-Newton method）也是求解无约束最优化问题的常用方法，有收敛速度快的优点。牛顿法是迭代算法，每一步需要求解目标函数的黑塞矩阵的逆矩阵，计算比较复杂。拟牛顿法通过正定矩阵近似黑塞矩阵的逆矩阵或黑塞矩阵，简化了这一计算过程。

1. 牛顿法

考虑无约束最优化问题

$$\min_{x\in\mathbf{R}^n} f(x) \tag{B.1}$$

其中 x^* 为目标函数的极小点。

假设 $f(x)$ 具有二阶连续偏导数，若第 k 次迭代值为 $x^{(k)}$，则可将 $f(x)$ 在 $x^{(k)}$ 附近进行二阶泰勒展开：

$$f(x)=f(x^{(k)})+g_k^{\mathrm{T}}(x-x^{(k)})+\frac{1}{2}(x-x^{(k)})^{\mathrm{T}}H(x^{(k)})(x-x^{(k)}) \tag{B.2}$$

这里，$g_k=g(x^{(k)})=\nabla f(x^{(k)})$ 是 $f(x)$ 的梯度向量在点 $x^{(k)}$ 的值，$H(x^{(k)})$ 是 $f(x)$ 的黑塞矩阵（Hessian matrix）

$$H(x)=\left[\frac{\partial^2 f}{\partial x_i\partial x_j}\right]_{n\times n} \tag{B.3}$$

在点 $x^{(k)}$ 的值。函数 $f(x)$ 有极值的必要条件是在极值点处一阶导数为 0，即梯度向量为 0。特别是当 $H(x^{(k)})$ 是正定矩阵时，函数 $f(x)$ 的极值为极小值。

牛顿法利用极小点的必要条件

$$\nabla f(x)=0 \tag{B.4}$$

每次迭代中从点 $x^{(k)}$ 开始，求目标函数的极小点，作为第 $k+1$ 次迭代值 $x^{(k+1)}$。具体地，假设 $x^{(k+1)}$ 满足：

$$\nabla f(x^{(k+1)})=0 \tag{B.5}$$

由式 (B.2) 有

$$\nabla f(x)=g_k+H_k(x-x^{(k)}) \tag{B.6}$$

其中 $H_k=H(x^{(k)})$。这样，式 (B.5) 成为

$$g_k+H_k(x^{(k+1)}-x^{(k)})=0 \tag{B.7}$$

因此，

$$x^{(k+1)}=x^{(k)}-H_k^{-1}g_k \tag{B.8}$$

或者

$$x^{(k+1)}=x^{(k)}+p_k \tag{B.9}$$

其中，

$$H_kp_k=-g_k \tag{B.10}$$

用式 (B.8) 作为迭代公式的算法就是牛顿法。

算法 B.1（牛顿法）

输入：目标函数 $f(x)$，梯度 $g(x)=\nabla f(x)$，黑塞矩阵 $H(x)$，精度要求 ε；

输出：$f(x)$ 的极小点 x^*。

（1）取初始点 $x^{(0)}$，置 $k=0$。

（2）计算 $g_k=g(x^{(k)})$。

（3）若 $\|g_k\|<\varepsilon$，则停止计算，得近似解 $x^*=x^{(k)}$。

（4）计算 $H_k=H(x^{(k)})$，并求 p_k

$$H_kp_k=-g_k$$

（5）置 $x^{(k+1)}=x^{(k)}+p_k$。

（6）置 $k=k+1$，转（2）。 ■

步骤（4）求 p_k，$p_k=-H_k^{-1}g_k$，要求 H_k^{-1}，计算比较复杂，所以有其他改进的方法。

2. 拟牛顿法的思路

在牛顿法的迭代中，需要计算黑塞矩阵的逆矩阵 H^{-1}，这一计算比较复杂，考虑用一个 n 阶矩阵 $G_k = G(x^{(k)})$ 来近似代替 $H_k^{-1} = H^{-1}(x^{(k)})$。这就是拟牛顿法的基本想法。

先看牛顿法迭代中黑塞矩阵 H_k 满足的条件。首先，H_k 满足以下关系。在式 (B.6) 中取 $x = x^{(k+1)}$，即得

$$g_{k+1} - g_k = H_k(x^{(k+1)} - x^{(k)}) \tag{B.11}$$

记 $y_k = g_{k+1} - g_k$，$\delta_k = x^{(k+1)} - x^{(k)}$，则

$$y_k = H_k\delta_k \tag{B.12}$$

或

$$H_k^{-1}y_k = \delta_k \tag{B.13}$$

式 (B.12) 或式 (B.13) 称为拟牛顿条件。

如果 H_k 是正定的（H_k^{-1} 也是正定的），那么可以保证牛顿法搜索方向 p_k 是下降方向。这是因为搜索方向是 $p_k = -H_k^{-1}g_k$，由式 (B.8) 有

$$x = x^{(k)} + \lambda p_k = x^{(k)} - \lambda H_k^{-1}g_k \tag{B.14}$$

所以 $f(x)$ 在 $x^{(k)}$ 的泰勒展开式 (B.2) 可以近似写成：

$$f(x) = f(x^{(k)}) - \lambda g_k^{\mathrm{T}}H_k^{-1}g_k \tag{B.15}$$

因 H_k^{-1} 正定，故有 $g_k^{\mathrm{T}}H_k^{-1}g_k > 0$。当 λ 为一个充分小的正数时，总有 $f(x) < f(x^{(k)})$，也就是说 p_k 是下降方向。

拟牛顿法将 G_k 作为 H_k^{-1} 的近似，要求矩阵 G_k 满足同样的条件。首先，每次迭

3. DFP (Davidon-Fletcher-Powell) 算法 (DFP algorithm)

DFP 算法选择 G_{k+1} 的方法是，假设每一步迭代中矩阵 G_{k+1} 是由 G_k 加上两个附加项构成的，即

$$G_{k+1} = G_k + P_k + Q_k \tag{B.18}$$

其中 P_k，Q_k 是待定矩阵。这时，

$$G_{k+1}y_k = G_ky_k + P_ky_k + Q_ky_k \tag{B.19}$$

为使 G_{k+1} 满足拟牛顿条件，可使 P_k 和 Q_k 满足：

$$P_ky_k = \delta_k \tag{B.20}$$

$$Q_ky_k = -G_ky_k \tag{B.21}$$

事实上，不难找出这样的 P_k 和 Q_k，例如取

$$P_k = \frac{\delta_k\delta_k^{\mathrm{T}}}{\delta_k^{\mathrm{T}}y_k} \tag{B.22}$$

$$Q_k = -\frac{G_ky_ky_k^{\mathrm{T}}G_k}{y_k^{\mathrm{T}}G_ky_k} \tag{B.23}$$

这样就可得到矩阵 G_{k+1} 的迭代公式：

$$G_{k+1} = G_k + \frac{\delta_k\delta_k^{\mathrm{T}}}{\delta_k^{\mathrm{T}}y_k} - \frac{G_ky_ky_k^{\mathrm{T}}G_k}{y_k^{\mathrm{T}}G_ky_k} \tag{B.24}$$

称为 DFP 算法。

可以证明，如果初始矩阵 G_0 是正定的，则迭代过程中的每个矩阵 G_k 都是正定的。

DFP 算法如下：

算法 B.2（DFP 算法）

输入：目标函数 $f(x)$，梯度 $g(x) = \nabla f(x)$，精度要求 ε；

输出：$f(x)$ 的极小点 x^*。

(1) 选定初始点 $x^{(0)}$，取 G_0 为正定对称矩阵，置 $k = 0$。

(2) 计算 $g_k = g(x^{(k)})$。若 $\|g_k\| < \varepsilon$，则停止计算，得近似解 $x^* = x^{(k)}$；否则转 (3)。

(3) 置 $p_k = -G_kg_k$。

(4) 一维搜索：求 λ_k 使得

$$f(x^{(k)} + \lambda_kp_k) = \min_{\lambda \geqslant 0} f(x^{(k)} + \lambda p_k)$$

(5) 置 $x^{(k+1)} = x^{(k)} + \lambda_kp_k$。

4. BFGS (Broyden-Fletcher-Goldfarb-Shanno) 算

BFGS 算法是最流行的拟牛顿算法。

可以考虑用 G_k 逼近黑塞矩阵的逆矩阵 H^{-1}，也

这时，相应的拟牛顿条件是

$$B_{k+1}\delta_k = y_k$$

可以用同样的方法得到另一迭代公式。首先令

$$B_{k+1} = B_k + P_k +$$

$$B_{k+1}\delta_k = B_k\delta_k + P_k\delta_k$$

考虑使 P_k 和 Q_k 满足：

$$P_k\delta_k = y_k$$

$$Q_k\delta_k = -B_k\delta_k$$

找出适合条件的 P_k 和 Q_k，得到 BFGS 算法矩阵 B

$$B_{k+1} = B_k + \frac{y_ky_k^{\mathrm{T}}}{y_k^{\mathrm{T}}\delta_k} - \frac{B_k}{\delta}$$

可以证明，如果初始矩阵 B_0 是正定的，则迭代过

下面写出 BFGS 拟牛顿算法。

算法 B.3（BFGS 算法）

输入：目标函数 $f(x)$，$g(x) = \nabla f(x)$，精度要求

输出：$f(x)$ 的极小点 x^*。

(1) 选定初始点 $x^{(0)}$，取 B_0 为正定对称矩阵，

(2) 计算 $g_k = g(x^{(k)})$。若 $\|g_k\| < \varepsilon$，则停止计算

- **Gradient Descent:**
  Compute true gradient exactly from all N examples
- **Stochastic Gradient Descent (SGD):**
  Approximate true gradient by the gradient of one randomly chosen example
- **Mini-Batch SGD:**
  Approximate true gradient by the average gradient of K randomly chosen examples

while not converged: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \mathbf{g}$

Three variants of first-order optimization:

$$\text{Gradient Descent: } \mathbf{g} = \nabla J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla J^{(i)}(\boldsymbol{\theta})$$

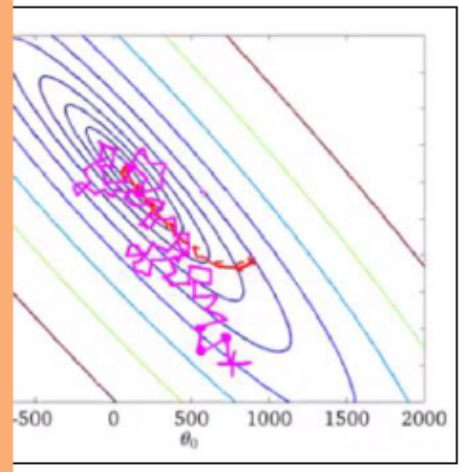
$$\text{SGD: } \mathbf{g} = \nabla J^{(i)}(\boldsymbol{\theta}) \quad \text{where } i \text{ sampled uniformly}$$

$$\text{Mini-batch SGD: } \mathbf{g} = \frac{1}{S} \sum_{s=1}^{S} \nabla J^{(i_s)}(\boldsymbol{\theta}) \quad \text{where } i_s \text{ sampled uniformly } \forall s$$

Stochastic Gradient Descent (SGD)

Algorithm 2 Stochastic Gradient Descent (SGD)

1: **procedure** SGD($\mathcal{D}, \boldsymbol{\theta}^{(0)}$)
2: $\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^{(0)}$
3: $\quad$**while** not converged **do**
4: $\quad\quad$**for** $i \in$ shuffle($\{1, 2, \ldots, N\}$) **do**
5: $\quad\quad\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J^{(i)}(\boldsymbol{\theta})$
6: $\quad$**return** $\boldsymbol{\theta}$



We need a per-example objective:

$$\text{Let } J(\boldsymbol{\theta}) = \sum_{i=1}^{N} J^{(i)}(\boldsymbol{\theta})$$

In practice, it is common to implement SGD using sampling **without** replacement (i.e. shuffle({1,2,… N}), even though most of the theory is for sampling **with** replacement (i.e. Uniform({1,2,… N}).

Many other algorithms

- **Conjugate Gradient Method**
- **Modified Newton's Method**
- **Quasi-Newton Methods (拟牛顿)**
  - …
  - Davidon-Fletcher-Powell (**DFP**) Method
  - Broyden-Fletcher-Goldfarb-Shanno (**BFGS**) Method
    - The DFP update was soon superseded by the BFGS formula, which is generally considered to be the most effective quasi-Newton update.

Broyden, Fletcher, Goldfarb and Shanno at the NATO Optimization Meeting (Cambridge, UK, 1983), a seminal meeting for continuous optimization

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

https://cdn.ttgtmedia.com/searchSystemsChannel/downloads/Oracle_Performance_Survival_Guide_ch13.pdf

Chapter 13

PARALLEL SQL

Parallel SQL enables a SQL statement to be processed by multiple threads or processes simultaneously.

Today's widespread use of dual and quad core processors means that even the humblest of modern computers running an Oracle database will contain more than one CPU. Although desktop and laptop computers might have only a single disk device, database server systems typically have database files spread—striped—across multiple, independent disk devices. Without parallel technology—when a SQL statement is processed in *serial*—a session can make use of only one of these CPUs or disk devices at a time. Consequently, serial execution of a SQL statement cannot make use of all the processing power of the computer. Parallel execution enables a single session and SQL statement to harness the power of multiple CPU and disk devices.

Parallel processing can improve the performance of suitable SQL statements to a degree that is often not possible by any other method. Parallel processing is available in Oracle Enterprise Edition only.

In this chapter we look at how Oracle can parallelize SQL statements and how you can use this facility to improve the performance of individual SQLs or the application as a whole.

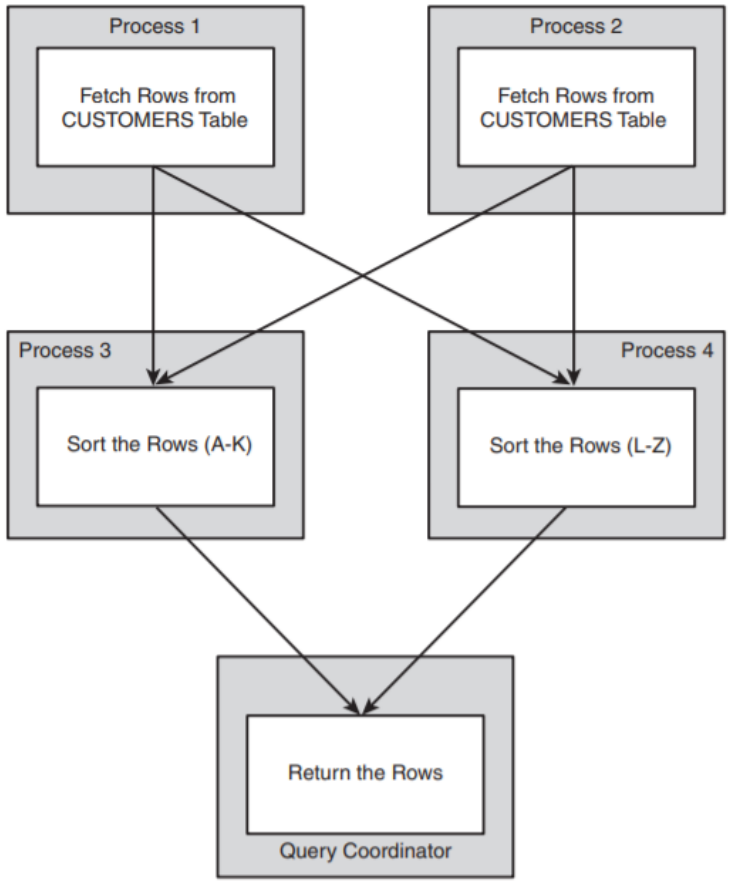
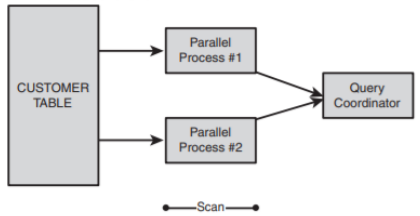
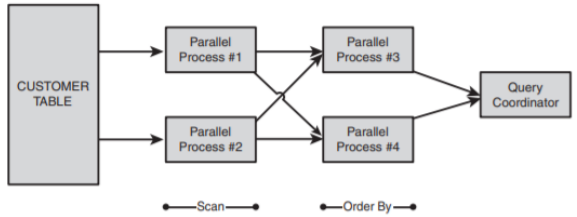


FIGURE 13-2 Parallel Execution.


```
SELECT /*+ parallel(c,2) */ *
  FROM customers c
```




```
SELECT /*+ parallel(c,2) */ *
  FROM customers c
 ORDER BY cust_last_name,cust_first_name
```




```
SELECT /*+ parallel(c,2) */ cust_last_name,count(*)
  FROM customers c
 GROUP BY cust_last_name
 ORDER BY 2 desc
```

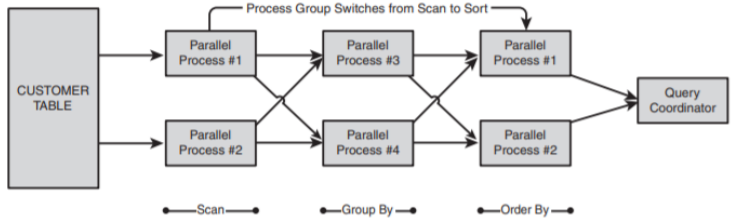
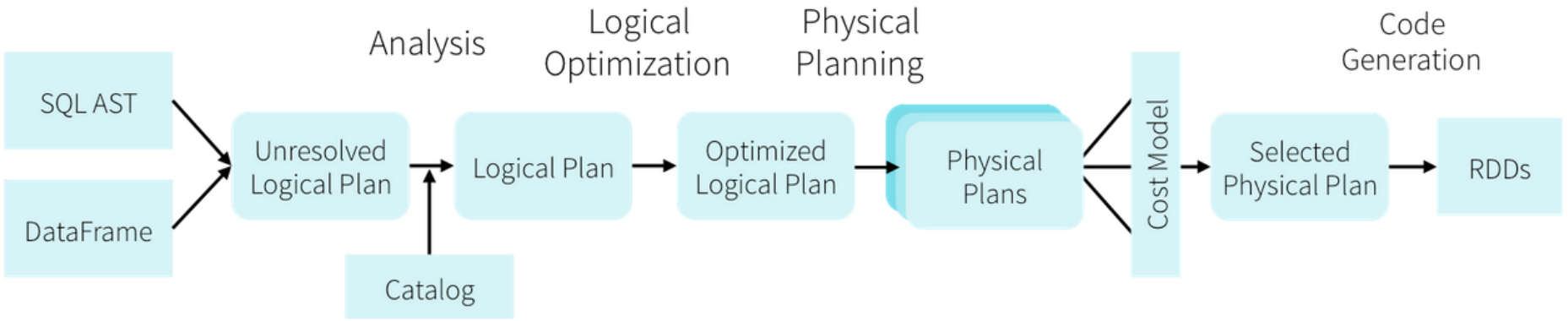


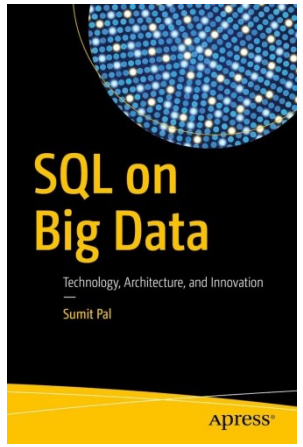
FIGURE 13-3 Parallel process allocation for a DOP of 2.

Challenges of SQL in Big Data

- Share similar flowchart



- But, Data are scattered in different nodes
  - Reducing the amount of data to be shuffled is a major challenge
  - How to optimize?
  - How to carry out the execution efficiently?
  - How to process SQL queries on streaming data?
  - …



https://ieeexplore.ieee.org/document/8611821

Efficiently Running SQL Queries on GPU

Publisher: IEEE

Cite This PDF

Dimitri Dojchinovski ; Marjan Gusev ; Vladimir Zdraveski All Authors

Abstract

Document Sections

I. Introduction

II. Related Work

III. Implementation

IV. Results and Comparison

V. Conclusion

Abstract:
In the last decade parallel computation (CUDA in particular), has accelerated the world. Applications used in astronomy, biology, chemistry, physics, data mining, manufacturing, finance, machine-learning and other computational intense fields are increasingly using CUDA to deliver the benefits of GPU acceleration. Also, CUDA is used in boosting the performance of database-oriented problems, speeding up the SQL performance on a huge dataset resulting in acquiring the data in rapid speed. This paper focuses on reading data from a. csv file, implementing kernels and performing the basic SQL aggregation functions on the GPU, and comparing their performance with the MySQL counterpart, showing the worthiness and importance of the concept.

Published in: 2018 26th Telecommunications Forum (TELFOR)

More Like This

Quality-Based SQL: Specifying Information Quality in Relational Database Queries
Computer
Published: 2015

Query processing over data warehouse using relational databases and NoSQL
2012 XXXVIII Conferencia Latinoamericana En Informatica (CLEI)
Published: 2012

Feedback

10 Other Computation examples

- Shortest Path in a Weighted Diagraph
- Matrix Multiplication
- FFT
- MRI
- N-Body
- N-S
- IR – PageRanking
- Optimization
- SQL
- NWP, Ocean, …

数值海洋与大气模式（三）：从0到1实现浅水(波)模式

文件(F) 编辑(E) 段落(P) 格式(O) 视图(V) 主题(T) 帮助(H)

文件 大纲

- 前言
- 资料
- 【泛函分析学习笔记10】Banach空间与Cauchy列
- 通过差分法来近似导数（偏导）
- 并行计算：介绍
- 求解 PDE 方程组
- How_to_Solve_Coupled_Differential_Equations_ODEs_in_Python-MXUMJMrX2Gw.mp4
- 【HSandip Mazumder】SIMPLE算法-偏微分方程的数值求解方法 - 是FVM方法，作罢 (2024年3月6日14:44:29)
- 数值海洋与大气模式（一）：控制方程和差分离散化
- ++ 数值海洋与大气模式（二）：从0到1实现浅水模式（上）
  - 一维浅水模式
  - 版权声明
  - 参考书目
- **++ 数值海洋与大气模式（三）：从0到1实现浅水模式（下）**
  - 二维浅水模式
  - 版权声明
  - 参考书目
- 数值海洋与大气模式（四）：POM模式框架
- 数值海洋与大气模式（五）：湍流模式与参数化方案
- 数值海洋与大气模式（六）：资料同化——逐步订正法与最优插值法概述
- 数值海洋与大气模式（七）：变分原理与泛函分析在模式中的应用初步

"只要把y和f理解为向量，那么，前面所提供的各种计算公式即可用于一阶方程组的情形"。欧拉、龙格-库塔计算

++ 数值海洋与大气模式（三）：从0到1实现浅水模式（下）

https://zhuanlan.zhihu.com/p/348231138

二维浅水模式

上文讨论了浅水方程在一维情况下的求解，本文探讨浅水方程在二维情况下的求解，并考虑加上底摩擦和风应力的情况。

首先考虑最简单的二维浅水方程，形式如下。

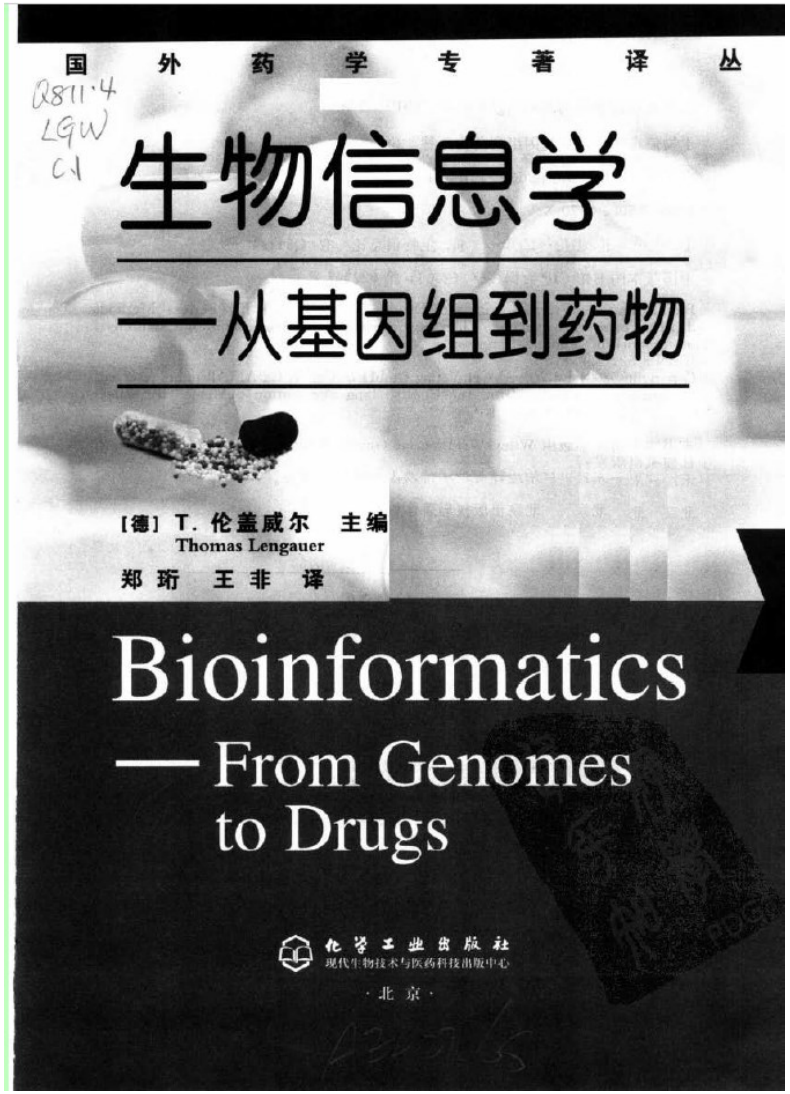
$$\frac{\partial u}{\partial t} = -g\frac{\partial \eta}{\partial x}$$
$$\frac{\partial v}{\partial t} = -g\frac{\partial \eta}{\partial y}$$
$$\frac{\partial \eta}{\partial t} = -\frac{\partial (uh)}{\partial x} - \frac{\partial (vh)}{\partial y}$$

还是同样的过程，对其做离散化，得到下式。

$$u_{j,k}^{n+1} = u_{j,k}^{n} - \Delta t g\left(\eta_{j,k+1}^{n} - \eta_{j,k}^{n}\right)/\Delta x$$
$$v_{j,k}^{n+1} = v_{j,k}^{n} - \Delta t g\left(\eta_{j+1,k}^{n} - \eta_{j,k}^{n}\right)/\Delta y$$
$$\eta_{j,k}^{*} = \eta_{j,k}^{n} - \Delta t\left\{\left(u_{j,k}^{n+1}h_e - u_{j,k-1}^{n+1}h_w\right)/\Delta x - \left(v_{j,k}^{n+1}h_n - v_{j-1,k}^{n+1}h_s\right)/\Delta y\right\}$$

上文提到，为了方便实现中央差分来取得二阶精度，选择了交错网格，使得 η 和 u 总是相差半个网格距离。在二维的情况，速度是 u

23 / 19501 词



- 生物信息学：从基因组到药物
- [德] 轮盖威尔 (T. Lengauer). 王非 译.
- 生物信息学：从基因组到药物.
- 北京：化学工业出版社.
- 2006.

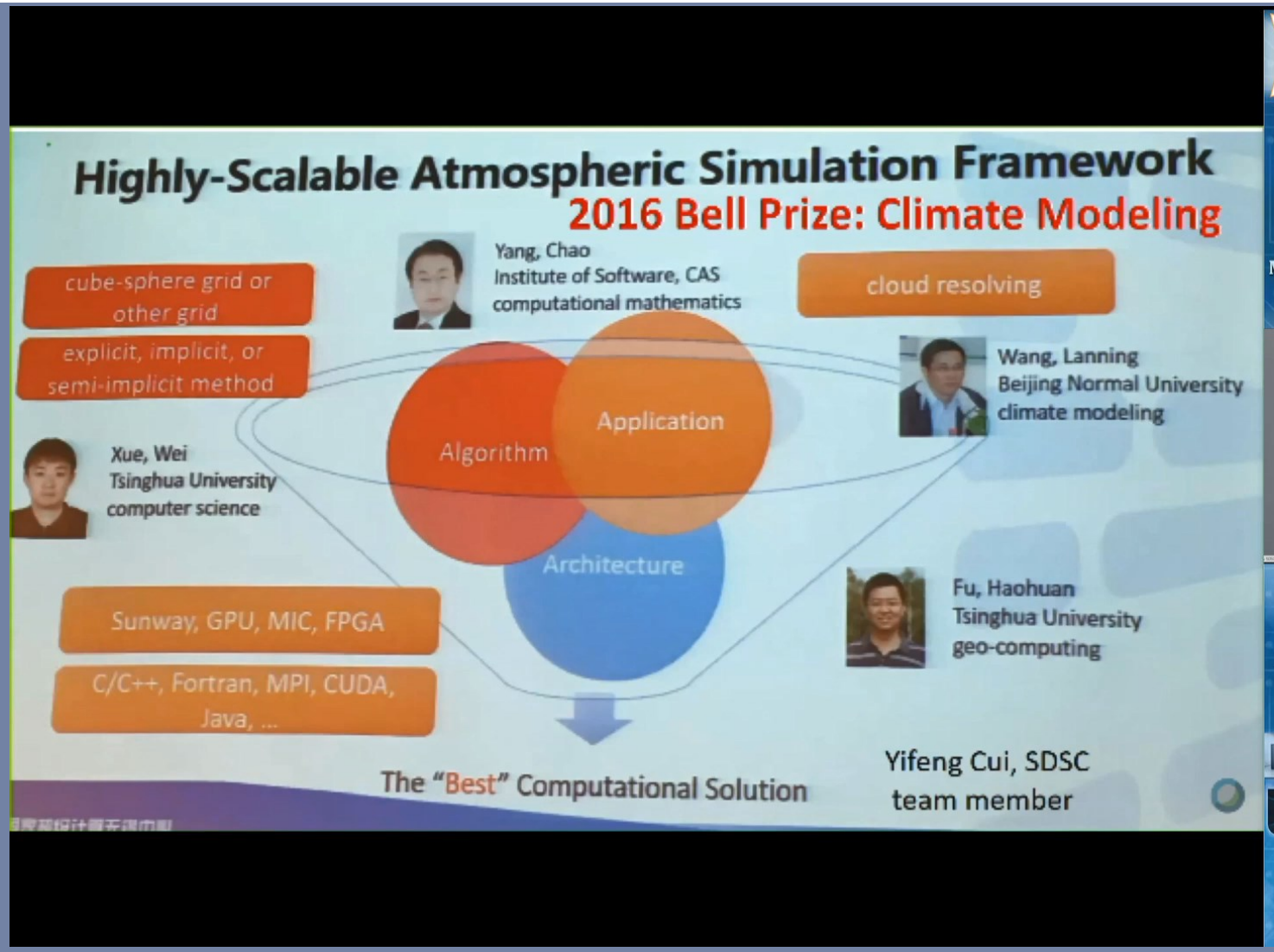
Gordon Bell Prizes: Science at Scale

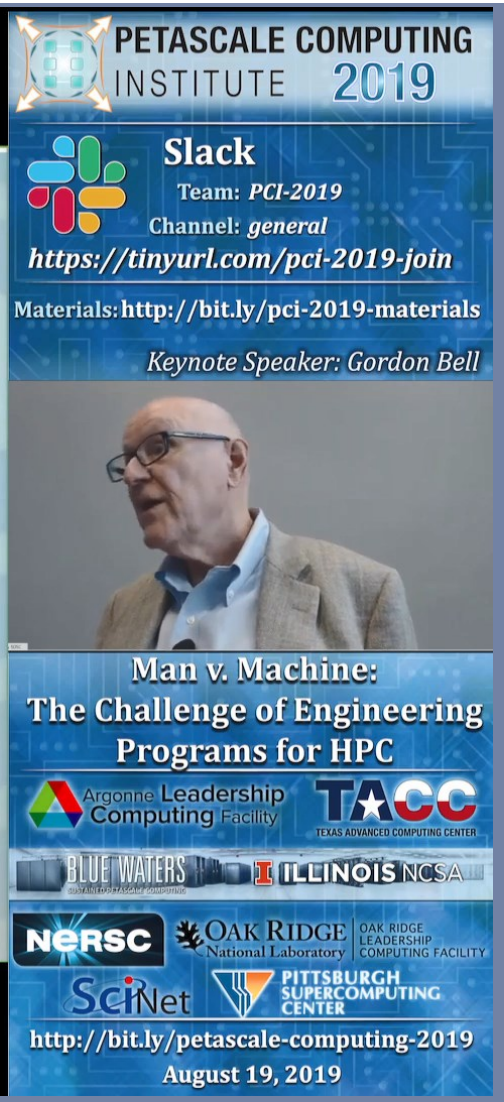
Established in 1987 with a cash award of $10,000 (since 2011), funded by Gordon Bell, a pioneer in HPC.
For innovation in applying *HPC to applications in science, engineering, and data analytics*.

China 1st wined Gordon 2016

- **A Chinese team on Friday won the 2016 ACM Gordon Bell prize, a top honor in high-performance computing, for an application running on China's fastest supercomputer.**
  - It is the first time a Chinese team has won the award.

The project, named "10M-Core Scalable Fully-Implicit Solver for Nonhydrostatic Atmospheric Dynamics," presents a method for calculating atmospheric dynamics, according to the Association for Computing Machinery, which presented the award at the International Supercomputing Conference in Salt Lake City in the United States


Highly-Scalable Atmospheric Simulation Framework
2016 Bell Prize: Climate Modeling
Yang, Chao
Institute of Software, CAS
computational mathematics
cube-sphere grid or other grid
explicit, implicit, or semi-implicit method
cloud resolving
Wang, Lanning
Beijing Normal University
climate modeling
Xue, Wei
Tsinghua University
computer science
Algorithm
Application
Architecture
Sunway, GPU, MIC, FPGA
C/C++, Fortran, MPI, CUDA, Java, ...
Fu, Haohuan
Tsinghua University
geo-computing
The "Best" Computational Solution
Yifeng Cui, SDSC
team member

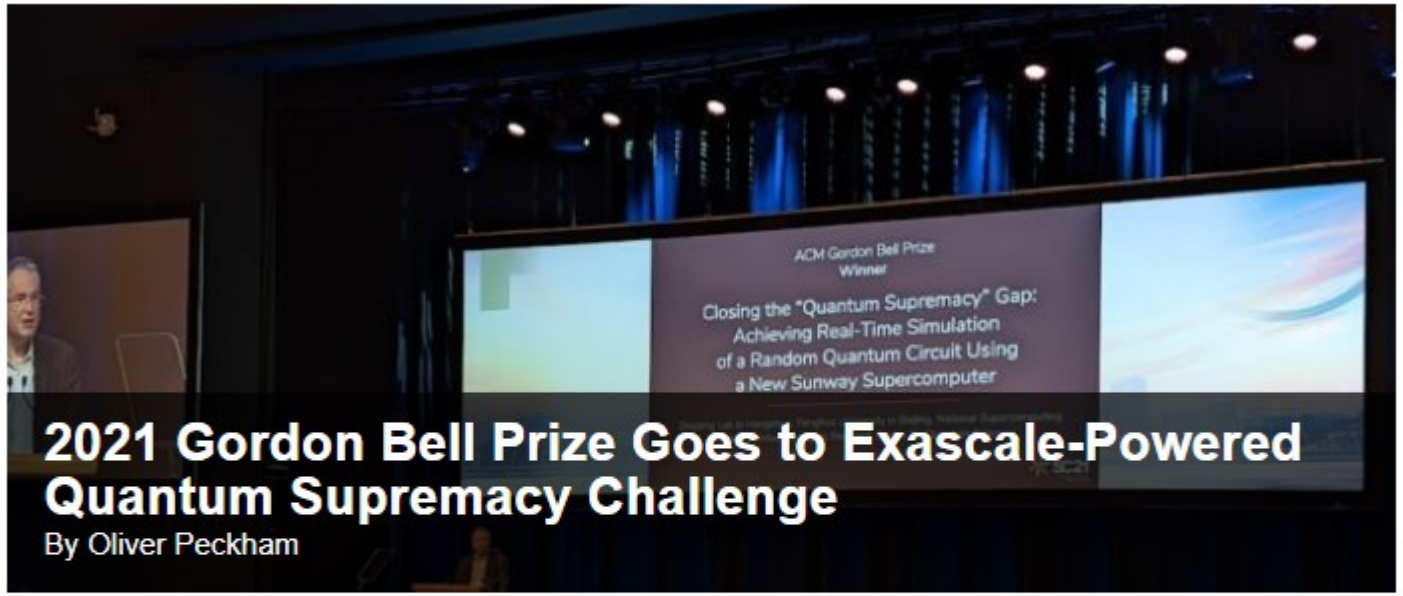

PETASCALE COMPUTING INSTITUTE 2019
Slack
Team: PCI-2019
Channel: general
https://tinyurl.com/pci-2019-join
Materials: http://bit.ly/pci-2019-materials
Keynote Speaker: Gordon Bell
Man v. Machine:
The Challenge of Engineering Programs for HPC
Argonne Leadership Computing Facility
TACC
TEXAS ADVANCED COMPUTING CENTER
BLUE WATERS
ILLINOIS NCSA
NERSC
OAK RIDGE National Laboratory
OAK RIDGE LEADERSHIP COMPUTING FACILITY
SciNet
PITTSBURGH SUPERCOMPUTING CENTER
http://bit.ly/petascale-computing-2019
August 19, 2019

- **2017 ACM Gordon Bell Prize awarded to Chinese team led by Tsinghua on Nonlinear Earthquake Simulation employing the world's fastest supercomputer**
  - 18.9-Pflops Nonlinear Earthquake Simulation on Sunway TaihuLight: Enabling Depiction of 18-Hz and 8-Meter Scenarios
- **2020年11月19日，全球高性能计算（HPC）最高奖——戈登贝尔奖（Gordon Bell Prize）正式揭晓。智源青年科学家、北京应用物理与计算数学研究所副研究员王涵所在团队，凭借其在“HPC+AI+第一性原理分子动力学”方向的工作：“Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning”，成功将本年度奖项收入囊中**
  - **这也是戈登贝尔奖历史上，中国团队第三次获奖。**
    - 智源学者中，杨超（时任中科院研究员，现为北京大学教授）团队在2016年也曾凭借“千万核可扩展大气动力学全隐式模拟”研究成果成为首支获得该奖的中国团队，一举打破美国、日本在该奖项上近30年的垄断，实现我国高性能计算应用戈登贝尔奖零突破

2021

SC21 St. Louis, MO | science & beyond.

https://www.hpcwire.com/2021/11/18/2021-gordon-bell-prize-goes-to-exascale-powered-quantum-supremacy-challenge/



2021 Gordon Bell Prize Goes to Exascale-Powered Quantum Supremacy Challenge

By Oliver Peckham

November 18, 2021

Today at the hybrid virtual/in-person SC21 conference, the organizers announced the winners of the 2021 ACM Gordon Bell Prize: a team of Chinese researchers leveraging the new exascale Sunway system to simulate quantum circuits.

Winner of the 2021 ACM Gordon Bell Prize

Closing the "Quantum Supremacy" Gap: Achieving Real-Time Simulation of a Random Quantum Circuit Using a New Sunway Supercomputer

Yong (Alexander) Liu, Xin (Lucy) Liu, Fang (Nancy) Li, Haohuan Fu, Yuling Yang, Jiawei Song, Pengpeng Zhao, Zhen Wang, Dajia Peng, Huarong Chen, Chu Guo, Heliang Huang, Wenzhao Wu and Dexun Chen

The fourteen researchers (whose affiliations span Zhejiang Lab, Tsinghua University, the National Supercomputing Center in Wuxi and the Shanghai Research Center for Quantum Sciences) leveraged the massive new Sunway exascale system that was more or less revealed during SC21 to conduct groundbreaking simulation of a quantum circuit.

"With Google's "Quantum Supremacy" declaration in 2019, stating that the Sycamore superconductive quantum computer is over a billion times faster than Summit (comparing 200 seconds against 10,000 years in the task of measuring/simulating one million samples), the dawn of the quantum age starts to unfold in a more affirmative way," the researchers wrote. "A later response from the IBM research team argues that they can accomplish the simulation on the classical Summit supercomputer … within a few days instead of 10,000 years."

https://awards.acm.org/bell/award-winners

戈登贝尔奖 [编辑]

维基百科，自由的百科全书

戈登贝尔奖（ACM Gordon Bell Prize）美国计算机协会设立于1987年，每年颁发，是一种超级电脑应用软件设计奖，奖金象征性1万美元。

戈登贝尔奖通常会由当年前500排行名列前茅的超级电脑系统之上所跑的的应用软件获得[1]，例如美国"泰坦"超级电脑、日本"京"超级电脑上的应用软件都曾得奖，从设立后30多年来都由美国和日本软件获得此奖，直到2016年中国打破此一规律[2]由神威·太湖之光超级电脑上的"全球大气非静力云分辨模拟"应用软件得奖。

奖项分为

- 最高性能奖 (Peak Performance)
- 最高性价比奖(Price/Performance)
- 特别奖 (Special Achievement)

外部链接 [编辑]

- Gordon Bell Prize - Award Winners: List By Year (页面存档备份，存于互联网档案馆)
- Gordon Bell Prize description from SC13
- ACM Gordon Bell Prize Winners 2006-present
- Earlier Prize Winners 1987–1999 (页面存档备份，存于互联网档案馆)

2022入围

首页 > 教学科研 > 正文

鹏程·神农入围戈登贝尔（Gordon Bell）新冠特别奖

日期：2022-11-20 12:57:29　信息工程学院、科研处　点击：115

11月17日，美国计算机协会（ACM）公布2022年度戈登贝尔新冠特别奖评选结果。北京大学深圳研究生院信息工程学院与鹏城实验室、山东大学组成的联合研究团队在自行研发的鹏程·神农生物信息研究平台上完成的“领先于病毒的进化——通过人工智能模拟预测未来高风险新冠病毒变异株”研究项目成功入围2022年度“戈登贝尔新冠特别奖”，也是本次入围唯一来自中国团队的项目。北大主要参与者是来自信息工程学院的田永鸿教授、陈杰副教授和博士研究生聂志伟和来自数学科学学院的杨超教授。该成果由美国华盛顿大学医院院长John Lynch教授、捷克查尔斯大学Martina Koziar Vasakova教授、西湖大学周强教授提名推荐。入围该奖的其余2个团队为：美国阿贡国家实验室、英伟达、芝加哥大学、加州理工学院联合团队及美国橡树岭国家实验室团队。鹏程·神农团队于众多世界级顶尖强队中脱颖而出，名列前茅，足见中国人工智能在计算集群和科研创新领域已处于全球顶尖水平。

鹏程·神农是基于“鹏城云脑II”超大规模算力集群和昇思MindSpore AI框架联合打造的面向生物医学领域的新一代数据密集型生命科学精准计算平台。该平台依托生物大数据、计算生物学理论和技术、人工智能算法和计算集群，实现新药创制和病毒演化预测。

团队研发了首个面向新冠病毒RBD区域变异的全环节模拟流程，通过多层次优化的计算策略、国际领先的新冠病毒变异体精准评价筛选算法，实现了对高风险变异株的演化模拟及精准预测。

为了在高维变异空间中实现高性能预测，团队充分融合专家知识，复刻病毒在真实世界中的变异规律，构建基于神农大模型的变异体生成器。生成的海量变异体通过多层次的精准病毒关键性质预测算法，进行高通量筛选，以模拟病毒在真实世界变异过程所面临的筛选压力，每秒可生成、筛选超百万条变异体，每天可生成、筛选超10^{11}条变异体。同时通过递进循环微调的范式，逐步缩小病毒的变异空间，最终实现病毒的全流程变异模拟。团队在两天内实现了新冠病毒Alpha、Beta、Gamma、Delta、Omicron BA.5等主流毒株的变异模拟，且可以准确预测大多数的高风险监测变异株，包括BF.7、BQ.1、BA.4.6等。

病毒变异不断冲击着人类抗疫战线。在新冠病毒新变种不断出现的情况下，对潜在高风险变异株的预测有助于疫苗和药物研发的提前部署，为疫情防控决策提供有力支撑。

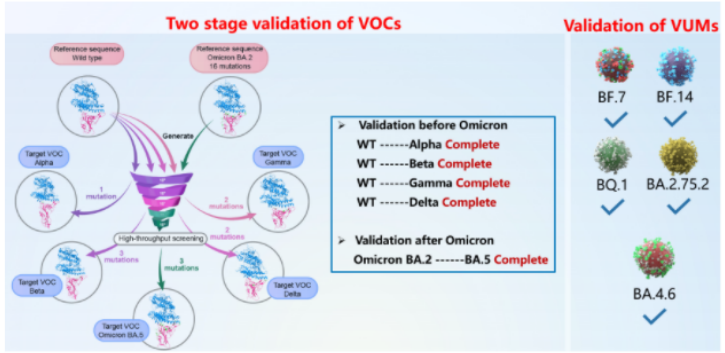
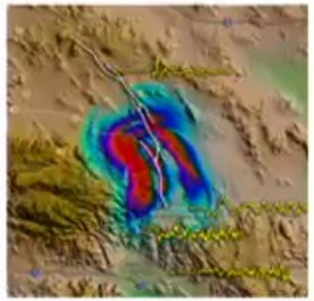


图3.神农AI大模型在两天内实现了对高风险变异株的演化模拟和精准预测

https://news.pkusz.edu.cn/info/1003/6838.htm

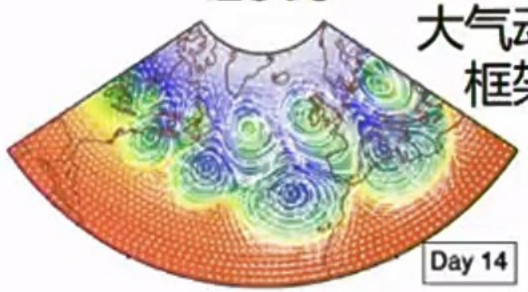
我国超算应用情况（以入围ACM Gordon Bell Prize为例）

2014

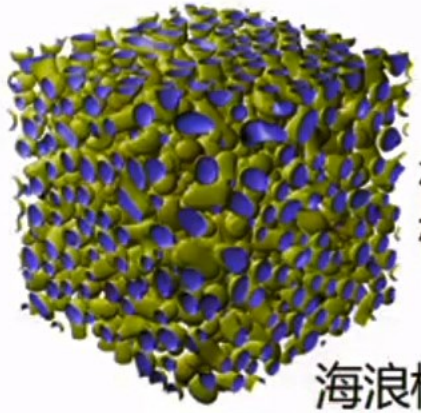


地震模拟

2016

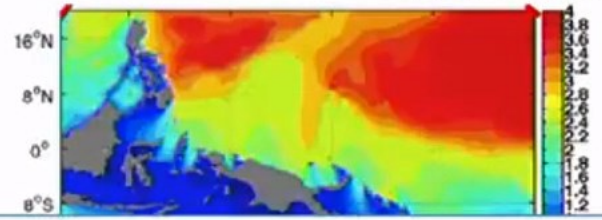


大气动力框架

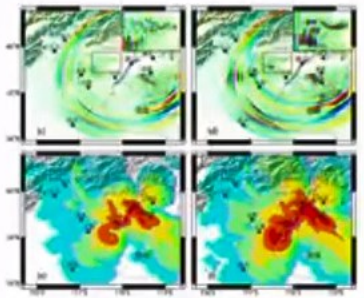


相场模拟

海浪模拟



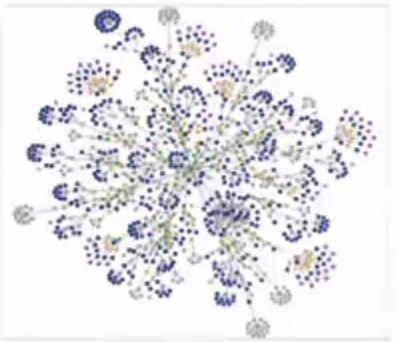
2017



地震模拟

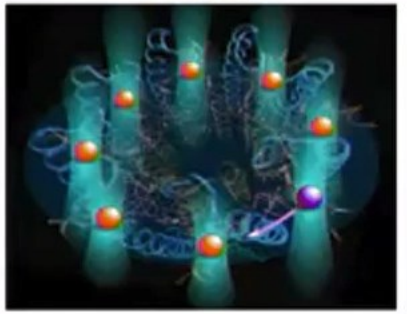
大气模拟

2018



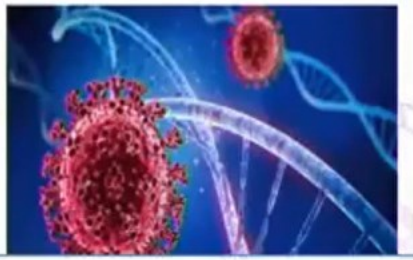
图计算框架

2021



量子模拟

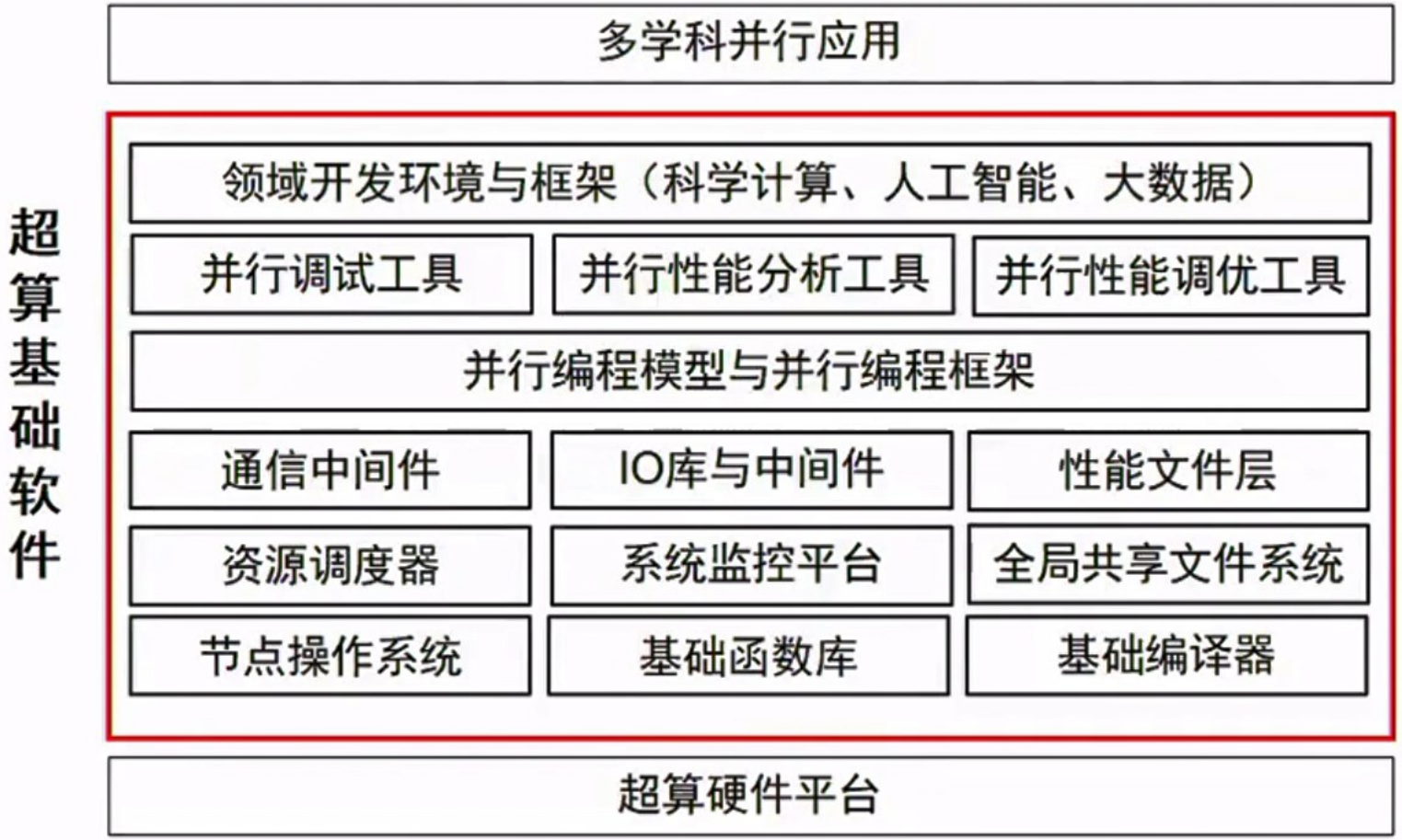
人造太阳



第一性原理

过去十年，依托我国顶尖超算系统，大规模并行应用设计和研制方面取得显著进步

超算基础软件是提升转化能力的关键之一



超算基础软件是实现并行应用开发、优化、部署、运行高效的基础和关键。

国产超算系统已经部署部分基础软件，仍有亟待解决的问题

- **科学计算与企业级应用的并行优化 (高性能计算技术丛书)**
- *刘文志 著*
- **2015**
- 机械工业出版社

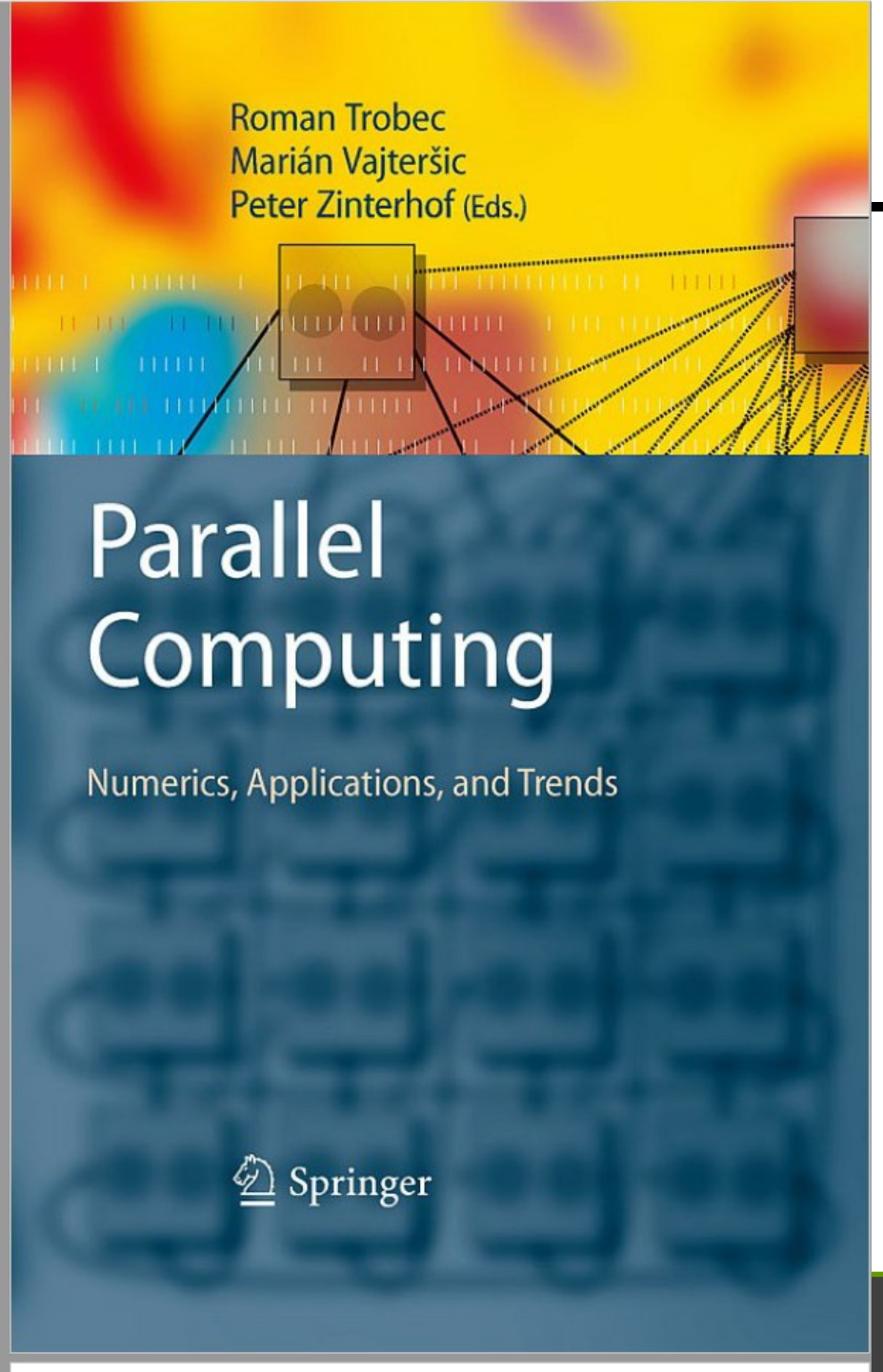
本系列的3本书相互之间有联系，也有其独立性：《并行算法设计与性能优化》介绍常见的串行代码优化方法和并行算法的设计；《并行编程方法与优化实践》介绍常见的向量化和并行编程环境及一些实例；《科学计算与企业级应用的并行优化》则介绍领域相关的算法与应用的性能优化。

目录

序
前言
第1章 多核向量处理器架构
1.1 众核系统结构
1.2 众核架构的一致性
1.3 多核向量处理器架构
1.3.1 Intel Haswell CPU架构
1.3.2 ARM A15多核向量处理器架构
1.3.3 AMD GCN GPU架构
1.3.4 NVIDIA Kepler和Maxwell GPU架构
1.4 Intel MIC架构
1.4.1 整体架构
1.4.2 计算单元
1.4.3 存储器单元
1.4.4 MIC架构上一些容易成为瓶颈的设计
1.5 OpenCL程序在多核向量处理器上的映射
1.5.1 OpenCL程序在多核向量CPU上的映射
1.5.2 OpenCL程序在NVIDIA GPU上的映射
1.5.3 OpenCL程序在AMD GCN上的映射
1.6 OpenCL程序在各众核硬件上执行的区别
1.7 众核编程模式
1.8 众核性能优化
1.9 MIC和GPU编程比较
1.10 本章小结
第2章 常见线性代数算法优化
2.1 稀疏矩阵与向量乘法
2.1.1 稀疏矩阵的存储格式
2.1.2 CSR格式稀疏矩阵与向量乘法
2.1.3 ELL格式稀疏矩阵与向量乘
2.2 对称矩阵与向量乘积
2.2.1 串行代码
2.2.2 向量化对称矩阵与向量乘积
2.2.3 OpenMP并行化
2.2.4 CUDA代码
2.3 三角线性方程组的解法
2.3.1 串行算法
2.3.2 串行算法优化
2.3.3 AVX优化实现
2.3.4 NEON优化实现
2.3.5 如何提高并行度
2.3.6 CUDA算法实现
2.4 矩阵乘法
2.4.1 AVX指令计算矩阵乘法
2.4.2 NEON指令计算矩阵乘法
2.4.3 GPU计算矩阵乘法
2.5 本章小结
第3章 优化偏微分方程的数值解法
3.1 热传递问题
3.1.1 C代码及性能
3.1.2 OpenMP代码及性能
3.1.3 OpenACC代码及性能
3.1.4 CUDA代码
3.2 简单三维Stencil
3.2.1 串行实现
3.2.2 Stencil在X86处理器上实现的困境
3.2.3 CUDA实现
3.3 本章小结
第4章 优化分子动力学算法
4.1 简单搜索的实现
4.1.1 串行代码
4.1.2 向量化实现分析
4.1.3 OpenMP实现
4.1.4 CUDA实现
4.2 范德华力计算
4.2.1 串行实现
4.2.2 向量化实现分析
4.2.3 OpenMP实现
4.2.4 CUDA实现
4.2.5 如何提高缓存的利用
4.3 键长伸缩力计算
4.3.1 串行实现
4.3.2 向量化实现
4.3.3 OpenMP实现
4.3.4 CUDA实现
4.4 径向分布函数计算
4.4.1 串行实现
4.4.2 向量化实现
4.4.3 OpenMP实现
4.4.4 CUDA实现
4.5 本章小结
第5章 机器学习算法
5.1 k-means算法
5.1.1 计算流程
5.1.2 计算元素所属分类
5.1.3 更新分类中心
5.1.4 入口函数
5.2 KNN算法
5.2.1 计算步骤
5.2.2 相似度计算
5.2.3 求前k个相似度最大元素
5.2.4 统计所属分类
5.3 二维卷积
5.3.1 X86实现
5.3.2 ARM实现
5.3.3 CUDA实现
5.4 四维卷积
5.4.1 X86实现
5.4.2 ARM实现
5.4.3 CUDA实现
5.5 多GPU并行优化深度
5.5.1 为什么要使用多
5.5.2 AlexNet示例
5.5.3 Caffe的主要计算
5.5.4 多GPU并行卷积
5.5.5 多GPU并行Caff
5.6 本章小结

书签

Preface
List of Contributors
Contents
About the Editors
1. Overview – Parallel Computing: Numerics, Applications, and Trends
2. Introduction to Parallel Computation
3. Tools for Parallel and Distributed Computing
4. Grid Computing
5. Parallel Structured Adaptive Mesh Refinement
6. Applications and Parallel Implementation of QMC Integration
7. Parallel Evolutionary Computation Framework for Single- and Multiobjective Optimization
8. WaLBerla: Exploiting Massively Parallel Systems for Lattice Boltzmann Simulations
9. Parallel Pseudo-Spectral Methods for the Time-Dependent Schrödinger Equation
10. Parallel Approaches in Molecular Dynamics Simulations
11. Parallel Computer Simulations of Heat Transfer in Biological Tissues
12. Parallel SVD Computing in the Latent Semantic Indexing Applications for Data Retrieval
13. Short-Vector SIMD Parallelization in Signal Processing
14. Financial Applications: Parallel Portfolio Optimization
15. The Future of Parallel Computation
Index




Michael W. Berry · Kyle A. Gallivan
Efstratios Gallopoulos · Ananth Grama
Bernard Philippe · Yousef Saad
Faisal Saied Editors
High-Performance Scientific Computing
Algorithms and Applications
Springer

Contents

1 **Parallel Numerical Computing from Illiac IV to Exascale—The Contributions of Ahmed H. Sameh** 1
Kyle A. Gallivan, Efstratios Gallopoulos, Ananth Grama, Bernard Philippe, Eric Polizzi, Yousef Saad, Faisal Saied, and Danny Sorensen

2 **Computational Capacity-Based Codesign of Computer Systems** . . . 45
David J. Kuck

3 **Measuring Computer Performance** 75
William Jalby, David C. Wong, David J. Kuck, Jean-Thomas Acquaviva, and Jean-Christophe Beyler

4 **A Compilation Framework for the Automatic Restructuring of Pointer-Linked Data Structures** 97
Harmen L.A. van der Spek, C.W. Mattias Holm, and Harry A.G. Wijshoff

5 **Dense Linear Algebra on Accelerated Multicore Hardware** 123
Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimire Tomov

6 **The Explicit Spike Algorithm: Iterative Solution of the Reduced System** 147
Carl Christian Kjelgaard Mikkelsen

7 **The Spike Factorization as Domain Decomposition Method; Equivalent and Variant Approaches** 157
Victor Eijkhout and Robert van de Geijn

8 **Parallel Solution of Sparse Linear Systems** 171
Murat Manguoglu

9 **Parallel Block-Jacobi SVD Methods** 185
Martin Bečka, Gabriel Okša, and Marián Vajteršic

10 **Robust and Efficient Multifrontal Solver for Large Discretized PDEs** 199
Jianlin Xia

11 **A Preconditioned Scheme for Nonsymmetric Saddle-Point Problems** 219
Abdelkader Baggag

12 **Effect of Ordering for Iterative Solvers in Structural Mechanics Problems** 251
Sami A. Kilic

13 **Scaling Hypre's Multigrid Solvers to 100,000 Cores** 261
Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang

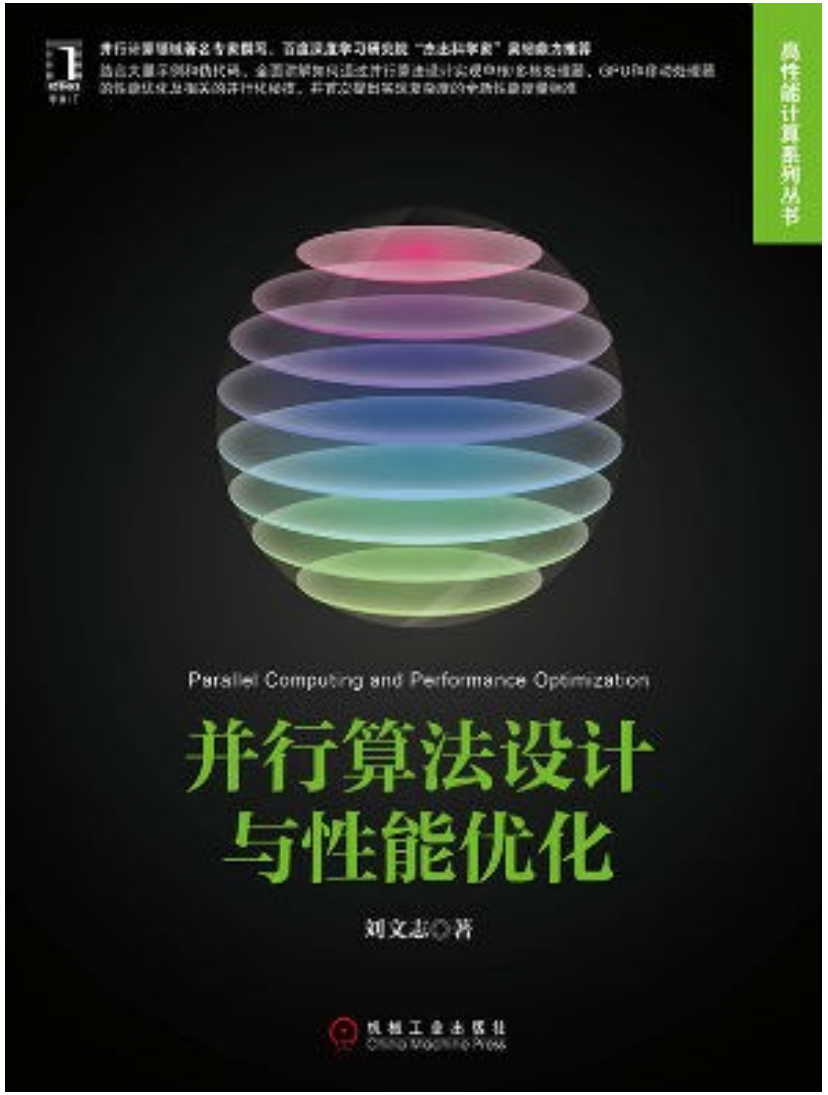
14 **A Riemannian Dennis-Moré Condition** 281
Kyle A. Gallivan, Chunhong Qi, and P.-A. Absil

15 **A Jump-Start of Non-negative Least Squares Solvers** 295
Mu Wang and Xiaoge Wang

16 **Fast Nonnegative Tensor Factorization with an Active-Set-Like Method** 311
Jingu Kim and Haesun Park

17 **Knowledge Discovery Using Nonnegative Tensor Factorization with Visual Analytics** 327
Andrey A. Puretskiy and Michael W. Berry

Index 343



整体而言，本书分为如下几个部分：

·理论基础，本部分主要介绍并行软件和硬件基础，并行算法设计思想以及一些软件优化方法。主要包括第1章、第2章、第3章、第5章。

·代码优化，本部分主要介绍常见的串行代码优化手段（不包括向量化）。主要内容是第4章。

·并行算法设计考量，本部分主要介绍如何设计优良的并行算法并将算法映射到硬件上。主要内容是第6章、第7章、第8章、第9章、第11章和第12章。

·如何将现有的串行代码并行化，主要内容是第10章。

第1章　主要介绍并行化和向量化的相关概念，如并行和向量化的作用、为什么并行化和向量化、并行或向量化面临的现实困难。另外还介绍了一些不写代码也能够利用多核处理器性能的一些方法。

第2章　介绍了现代处理器的特性，如指令级并行、向量化并行、线程级并行、处理器缓存金字塔、虚拟存储器和NUMA（非一致内存访问）。

第3章　介绍了算法性能和程序性能的度量与分析。算法性能分析和度量的主要标准是时间复杂度、空间复杂度和笔者自己提出的实现复杂度。程序性能的度量

第8章　介绍了并行

析了如何缓解某些缺点的

第9章　介绍了如何

uce、scan和流水线等并

读者能够通过模式解决一

第10章　介绍了并行

要注意的事项，最后以如

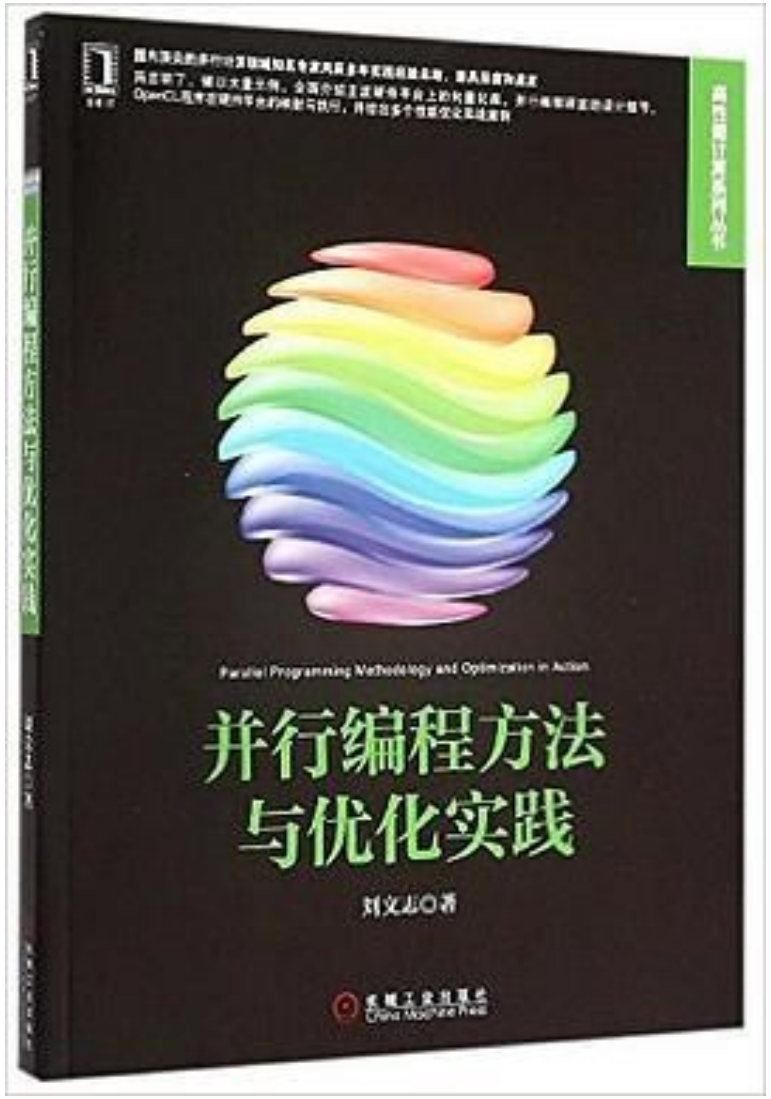
第11章　介绍了常见

超级并行模式下如何划分

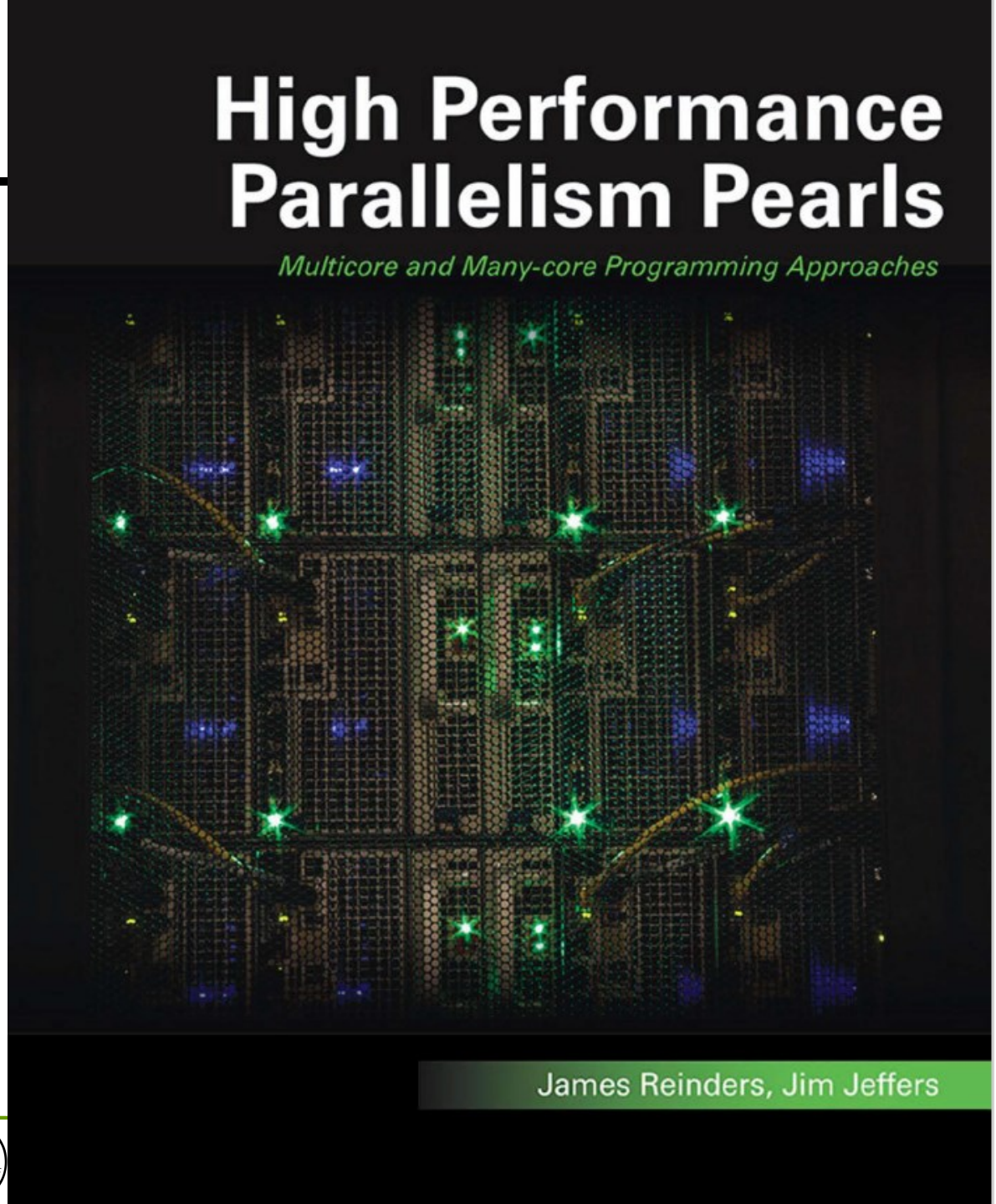
阵乘运算。

第12章　给出了设计

附录A　介绍了整数



- 并行编程方法与优化实践
- 刘文志
- 机械工业出版社
- 2015-06



High Performance
Parallelism Pearls
Multicore and Many-core
Programming Approaches

Contents

Contributors xv
Acknowledgments xxxix
Foreword xli
Preface xlv

CHAPTER 1 Introduction 1
Learning from Successful Experiences 1
Code Modernization 1
Modernize with Concurrent Algorithms 2
Modernize with Vectorization and Data Locality 2
Understanding Power Usage 2
ispc and OpenCL Anyone? 2
Intel Xeon Phi Coprocessor Specific 3
Many-Core, Neo-Heterogeneous 3
No "Xeon Phi" In The Title, Neo-Heterogeneous Programming 3
The Future of Many-Core 4
Downloads 4
For More Information 5

CHAPTER 2 From "Correct" to "Correct & Efficient": A Hydro2D Case Study with Godunov's Scheme 7
Scientific Computing on Contemporary Computers 7
Modern Computing Environments 8
CEA's Hydro2D 9
A Numerical Method for Shock Hydrodynamics 9
Euler's Equation 10
Godunov's Method 10
Where It Fits 12
Features of Modern Architectures 13
Performance-Oriented Architecture 13
Programming Tools and Runtimes 14
Our Computing Environments 14
Paths to Performance 15
Running Hydro2D 15
Hydro2D's Structure 15
Optimizations 20
Memory Usage 21
Thread-Level Parallelism 22

Arithmetic Efficiency and Instruction-Level Parallelism 30
Data-Level Parallelism 32
Summary 39
The Coprocessor vs the Processor 39
A Rising Tide Lifts All Boats 39
Performance Strategies 41
For More Information 42

CHAPTER 3 Better Concurrency and SIMD on HBM 43
The Application: HIROMB-BOOS-Model 43
Key Usage: DMI 44
HBM Execution Profile 44
Overview for the Optimization of HBM 45
Data Structures: Locality Done Right 46
Thread Parallelism in HBM 50
Data Parallelism: SIMD Vectorization 55
Trivial Obstacles 55
Premature Abstraction is the Root of All Evil 58
Results 61
Profiling Details 62
Scaling on Processor vs. Coprocessor 62
Contiguous Attribute 64
Summary 66
References 66
For More Information 66

CHAPTER 4 Optimizing for Reacting Navier-Stokes Equations 69
Getting Started 69
Version 1.0: Baseline 70
Version 2.0: ThreadBox 73
Version 3.0: Stack Memory 77
Version 4.0: Blocking 77
Version 5.0: Vectorization 80
Intel Xeon Phi Coprocessor Results 83
Summary 84
For More Information 85

CHAPTER 5 Plesiochronous Phasing Barriers 87
What Can Be Done to Improve the Code? 89
What More Can Be Done to Improve the Code? 91
Hyper-Thread Phalanx 91
What is Nonoptimal About This Strategy? 93

Coding the Hyper-Thread Phalanx 93
How to Determine Thread Binding to Core and HT Within Core? 94
The Hyper-Thread Phalanx Hand-Partitioning Technique 95
A Lesson Learned 97
Back to Work 99
Data Alignment 99
Use Aligned Data When Possible 100
Redundancy Can Be Good for You 100
The Plesiochronous Phasing Barrier 103
Let us do Something to Recover This Wasted Time 105
A Few "Left to the Reader" Possibilities 109
Xeon Host Performance Improvements Similar to Xeon Phi 110
Summary 115
For More Information 115

CHAPTER 6 Parallel Evaluation of Fault Tree Expressions 117
Motivation and Background 117
Expressions 117
Expression of Choice: Fault Trees 117
An Application for Fault Trees: Ballistic Simulation 118
Example Implementation 118
Using ispc for Vectorization 121
Other Considerations 126
Summary 128
For More Information 128

CHAPTER 7 Deep-Learning Numerical Optimization 129
Fitting an Objective Function 129
Objective Functions and Principle Components Analysis 134
Software and Example Data 135
Training Data 136
Runtime Results 139
Scaling Results 141
Summary 141
For More Information 142

CHAPTER 8 Optimizing Gather/Scatter Patterns 143
Gather/Scatter Instructions in Intel® Architecture 145
Gather/Scatter Patterns in Molecular Dynamics 145
Optimizing Gather/Scatter Patterns 148
Improving Temporal and Spatial Locality 148
Choosing an Appropriate Data Layout: AoS Versus SoA 150
On-the-Fly Transposition Between AoS and SoA 151
Amortizing Gather/Scatter and Transposition Costs 154
Summary 156
For More Information 157

CHAPTER 9 A Many-Core Implementation of the Direct N-Body Problem 159
N-Body Simulations 159
Initial Solution 159
Theoretical Limit 162
Reduce the Overheads, Align Your Data 164
Optimize the Memory Hierarchy 167
Improving Our Tiling 170
What Does All This Mean to the Host Version? 172
Summary 174
For More Information 174

CHAPTER 10 *N*-Body Methods 175
Fast *N*-Body Methods and Direct *N*-Body Kernels 175
Applications of *N*-Body Methods 176
Direct *N*-Body Code 177
Performance Results 179
Summary 182
For More Information 183

CHAPTER 11 Dynamic Load Balancing Using OpenMP 4.0 185
Maximizing Hardware Usage 185
The N-Body Kernel 187
The Offloaded Version 191
A First Processor Combined with Coprocessor Version 193
Version for Processor with Multiple Coprocessors 196
For More Information 200

CHAPTER 12 Concurrent Kernel Offloading 201
Setting the Context 201
Motivating Example: Particle Dynamics 202
Organization of This Chapter 203
Concurrent Kernels on the Coprocessor 204
Coprocessor Device Partitioning and Thread Affinity 204
Concurrent Data Transfers 210
Force Computation in PD Using Concurrent Kernel Offloading 213
Parallel Force Evaluation Using Newton's 3rd Law 213
Implementation of the Concurrent Force Computation 215
Performance Evaluation: Before and After 220

The Bottom Line 221
For More Information 223

CHAPTER 13 Heterogeneous Computing with MPI 225
MPI in the Modern Clusters 225
MPI Task Location 226
Single-Task Hybrid Programs 229
Selection of the DAPL Providers 231
The First Provider *OFA-V2-MLX4_0-1U* 231
The Second Provider ofa-v2-scif0 and the Impact of the Intra-Node Fabric 232
The Last Provider, Also Called the Proxy 232
Hybrid Application Scalability 234
Load Balance 236
Task and Thread Mapping 236
Summary 237
Acknowledgments 238
For More Information 238

CHAPTER 14 Power Analysis on the Intel® Xeon Phi™ Coprocessor 239
Power Analysis 101 239
Measuring Power and Temperature with Software 241
Creating a Power and Temperature Monitor Script 243
Creating a Power and Temperature Logger with the micsmc Tool 243
Power Analysis Using IPMI 245
Hardware-Based Power Analysis Methods 246
A Hardware-Based Coprocessor Power Analyzer 249
Summary 252
For More Information 253

CHAPTER 15 Integrating Intel Xeon Phi Coprocessors into a Cluster Environment 255
Early Explorations 255
Beacon System History 256
Beacon System Architecture 256
Hardware 256
Software Environment 256
Intel MPSS Installation Procedure 258
Preparing the System 258
Installation of the Intel MPSS Stack 259
Generating and Customizing Configuration Files 261
MPSS Upgrade Procedure 265
Setting Up the Resource and Workload Managers 265
TORQUE 265
Prologue 266
Epilogue 268
TORQUE/Coprocessor Integration 268
Moab 269
Improving Network Locality 269
Moab/Coprocessor Integration 269
Health Checking and Monitoring 269
Scripting Common Commands 271
User Software Environment 273
Future Directions 274
Summary 275
Acknowledgments 275
For More Information 275

CHAPTER 16 Supporting Cluster File Systems on Intel® Xeon Phi™ Coprocessors 277
Network Configuration Concepts and Goals 278
A Look At Networking Options 278
Steps to Set Up a Cluster Enabled Coprocessor 280
Coprocessor File Systems Support 281
Support for NFS 282
Support for Lustre® File System 282
Support for Fraunhofer BeeGFS® (formerly FHGFS) File System 284
Support for Panasas® PanFS® File System 285
Choosing a Cluster File System 285
Summary 285
For More Information 286

CHAPTER 17 NWChem: Quantum Chemistry Simulations at Scale 287
Introduction 287
Overview of Single-Reference CC Formalism 288
NWChem Software Architecture 291
Global Arrays 291
Tensor Contraction Engine 292
Engineering an Offload Solution 293
Offload Architecture 297
Kernel Optimizations 298
Performance Evaluation 301
Summary 304

Acknowledgments 305
For More Information 305

CHAPTER 18 Efficient Nested Parallelism on Large-Scale Systems 307
Motivation 307
The Benchmark 307
Baseline Benchmarking 309
Pipeline Approach—Flat_arena Class 310
Intel® TBB User-Managed Task Arenas 311
Hierarchical Approach—Hierarchical_arena Class 313
Performance Evaluation 314
Implication on NUMA Architectures 316
Summary 317
For More Information 318

CHAPTER 19 Performance Optimization of Black-Scholes Pricing 319
Financial Market Model Basics and the Black-Scholes Formula 320
Financial Market Mathematical Model 320
European Option and Fair Price Concepts 321
Black-Scholes Formula 322
Options Pricing 322
Test Infrastructure 323
Case Study 323
Preliminary Version—Checking Correctness 323
Reference Version—Choose Appropriate Data Structures 323
Reference Version—Do Not Mix Data Types 325
Vectorize Loops 326
Use Fast Math Functions: erff() vs. cdfnormf() 329
Equivalent Transformations of Code 331
Align Arrays 331
Reduce Precision if Possible 333
Work in Parallel 334
Use Warm-Up 334
Using the Intel Xeon Phi Coprocessor —"No Effort" Port 336
Use Intel Xeon Phi Coprocessor: Work in Parallel 337
Use Intel Xeon Phi Coprocessor and Streaming Stores 338
Summary 338
For More Information 339

CHAPTER 20 Data Transfer Using the Intel COI Library 341
First Steps with the Intel COI Library 341
COI Buffer Types and Transfer Performance 342
Applications 346
Summary 348
For More Information 348

CHAPTER 21 High-Performance Ray Tracing 349
Background 349
Vectorizing Ray Traversal 351
The Embree Ray Tracing Kernels 352
Using Embree in an Application 352
Performance 354
Summary 357
For More Information 358

CHAPTER 22 Portable Performance with OpenCL 359
The Dilemma 359
A Brief Introduction to OpenCL 360
A Matrix Multiply Example in OpenCL 364
OpenCL and the Intel Xeon Phi Coprocessor 366
Matrix Multiply Performance Results 368
Case Study: Molecular Docking 369
Results: Portable Performance 373
Related Work 374
Summary 375
For More Information 375

CHAPTER 23 Characterization and Optimization Methodology Applied to Stencil Computations 377
Introduction 377
Performance Evaluation 378
AI of the Test Platforms 379
AI of the Kernel 380
Standard Optimizations 382
Automatic Application Tuning 386
The Auto-Tuning Tool 392
Results 393
Summary 395
For More Information 395

CHAPTER 24 Profiling-Guided Optimization 397
Matrix Transposition in Computer Science 397
Tools and Methods 399
"Serial": Our Original In-Place Transposition 400
"Parallel": Adding Parallelism with OpenMP 405
"Tiled": Improving Data Locality 405

"Regularized": Microkernel with Multiversioning 411
"Planned": Exposing More Parallelism 417
Summary 421
For More Information 423

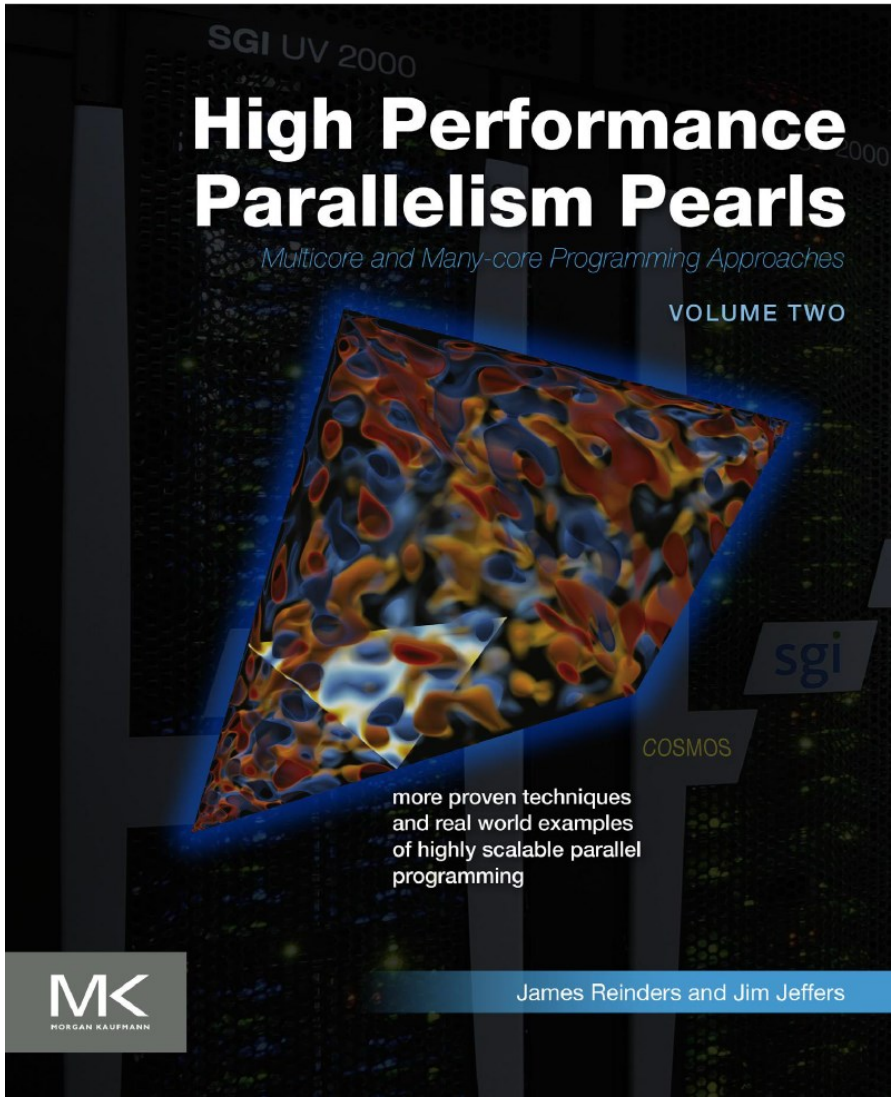
CHAPTER 25 Heterogeneous MPI Application Optimization with ITAC 425
Asian Options Pricing 425
Application Design 426
Synchronization in Heterogeneous Clusters 428
Finding Bottlenecks with ITAC 429
Setting Up ITAC 430
Unbalanced MPI Run 431
Manual Workload Balance 434
Dynamic "Boss-Workers" Load Balancing 436
Conclusion 439
For More Information 441

CHAPTER 26 Scalable Out-of-Core Solvers on a Cluster 443
Introduction 443
An OOC Factorization Based on ScaLAPACK 444
In-Core Factorization 445
OOC Factorization 446
Porting from NVIDIA GPU to the Intel Xeon Phi Coprocessor 447
Numerical Results 449
Conclusions and Future Work 454
Acknowledgments 454
For More Information 454

CHAPTER 27 Sparse Matrix-Vector Multiplication: Parallelization and Vectorization 457
Background 457
Sparse Matrix Data Structures 458
Compressed Data Structures 459
Blocking 462
Parallel SpMV Multiplication 462
Partially Distributed Parallel SpMV 462
Fully Distributed Parallel SpMV 463
Vectorization on the Intel Xeon Phi Coprocessor 465
Implementation of the Vectorized SpMV Kernel 467
Evaluation 470
On the Intel Xeon Phi Coprocessor 471
On Intel Xeon CPUs 472
Performance Comparison 474
Summary 474
Acknowledgments 475
For More Information 475

CHAPTER 28 Morton Order Improves Performance 477
Improving Cache Locality by Data Ordering 477
Improving Performance 477
Matrix Transpose 478
Matrix Multiply 482
Summary 488
For More Information 490

Author Index 491
Subject Index 495



Contents

Contributors xvii
Acknowledgments xliii
Foreword xlv
Preface xlix

CHAPTER 1 Introduction 1
Applications and techniques 1
SIMD and vectorization 2
OpenMP and nested parallelism 2
Latency optimizations 3
Python 3
Streams 3
Ray tracing 3
Tuning prefetching 3
MPI shared memory 4
Using every last core 4
OpenCL vs. OpenMP 4
Power analysis for nodes and clusters 4
The future of many-core 4
Downloads 5
For more information 5

CHAPTER 2 Numerical Weather Prediction Optimization 7
Numerical weather prediction: Background and motivation 7
WSM6 in the NIM 8
Shared-memory parallelism and controlling horizontal vector length 10
Array alignment 12
Loop restructuring 13
Compile-time constants for loop and array bounds 13
Performance improvements 19
Summary 22
For more information 23

CHAPTER 3 WRF Goddard Microphysics Scheme Optimization 25
The motivation and background 25
WRF Goddard microphysics scheme 26
Goddard microphysics scheme 26

Benchmark setup 26
Code optimization 30
Removal of the vertical dimension from temporary variables for a reduced memory footprint 31
Collapse i- and j-loops into smaller cells for smaller footprint per thread 32
Addition of vector alignment directives 35
Summary of the code optimizations 37
Analysis using an instruction Mix report 38
VTune performance metrics 38
Performance effects of the optimization of Goddard microphysics scheme on the WRF 39
Summary 39
Acknowledgments 40
For more information 40

CHAPTER 4 Pairwise DNA Sequence Alignment Optimization 43
Pairwise sequence alignment 43
Parallelization on a single coprocessor 44
Multi-threading using OpenMP 45
Vectorization using SIMD intrinsics 46
Parallelization across multiple coprocessors using MPI 48
Performance results 49
Summary 52
For more information 53

CHAPTER 5 Accelerated Structural Bioinformatics for Drug Discovery 55
Parallelism enables proteome-scale structural bioinformatics 56
Overview of *e*FindSite 57
Benchmarking dataset 57
Code profiling 58
Porting *e*FindSite for coprocessor offload 60
Parallel version for a multicore processor 65
Task-level scheduling for processor and coprocessor 66
Case study 70
Summary 70
For more information 72

CHAPTER 6 Amber PME Molecular Dynamics Optimization 73
Theory of MD 74
Acceleration of neighbor list building using the coprocessor 76

Acceleration of direct space sum using the coprocessor ... 77
Additional optimizations in coprocessor code ... 79
Removing locks whenever possible ... 79
Exclusion list optimization ... 79
Reduce data transfer and computation in offload code ... 81
Modification of load balance algorithm ... 82
PME direct space sum and neighbor list work ... 82
PME reciprocal space sum work ... 84
Bonded force work ... 85
Compiler optimization flags ... 85
Results ... 85
Conclusions ... 87
For more information ... 88

CHAPTER 7 Low-Latency Solutions for Financial Services Applications ... 91
Introduction ... 91
The opportunity ... 91
Packet processing architecture ... 93
The symmetric communication interface ... 94
Memory registration ... 95
Mapping remote memory via *scif_mmap()* ... 98
Optimizing packet processing on the coprocessor ... 99
Optimization #1: The right API for the job ... 99
Optimization #2: Benefit from write combining (WC) memory type ... 102
Optimization #3: "Pushing" versus "pulling" data ... 105
Optimization #4: "Shadow" pointers for efficient FIFO management ... 105
Optimization #5: Tickless kernels ... 107
Optimization #6: Single thread affinity and CPU "isolation" ... 108
Optimization #7: Miscellaneous optimizations ... 108
Results ... 109
Conclusions ... 110
For more information ... 111

CHAPTER 8 Parallel Numerical Methods in Finance ... 113
Overview ... 113
Introduction ... 113
Pricing equation for American option ... 114
Initial C/C++ implementation ... 115
Scalar optimization: Your best first step ... 116
Compiler invocation switches ... 116

Transcendental functions ... 119
Reuse as much as possible and reinvent as little as possible ... 119
Subexpression evaluation ... 120
SIMD parallelism—Vectorization ... 122
Define and use vector data ... 122
Vector arithmetic operations ... 123
Vector function call ... 124
Branch statements ... 124
Calling the vector version and the scalar version of the program ... 126
Vectorization by annotating the source code: #pragma SIMD ... 129
C/C++ vector extension versus #pragma SIMD ... 129
Thread parallelization ... 130
Memory allocation in NUMA system ... 132
Thread binding and affinity interface ... 133
Scale from multicore to many-core ... 133
Summary ... 136
For more information ... 136

CHAPTER 9 Wilson Dslash Kernel from Lattice QCD Optimization ... 139
The Wilson-Dslash kernel ... 140
Performance expectations ... 143
Refinements to the model ... 144
First implementation and performance ... 146
Running the naive code on Intel Xeon Phi coprocessor ... 149
Evaluation of the naive code ... 150
Optimized code: QPhiX and QphiX-Codegen ... 151
Data layout for vectorization ... 151
3.5D blocking ... 153
Load balancing ... 154
SMT threading ... 155
Lattice traversal ... 156
Code generation with QphiX-Codegen ... 157
QphiX-codegen code structure ... 159
Implementing the instructions ... 159
Generating Dslash ... 160
Prefetching ... 162
Generating the code ... 163
Performance results for QPhiX ... 164
Other benefits ... 166
The end of the road? ... 168
For more information ... 169

CHAPTER 10 Cosmic Microwave Background Analysis: Nested Parallelism in Practice 171
Analyzing the CMB with Modal 171
Optimization and modernization 174
Splitting the loop into parallel tasks 174
Introducing nested parallelism 178
Nested OpenMP parallel regions 179
OpenMP 4.0 teams 180
Manual nesting 181
Inner loop optimization 182
Results 184
Comparison of nested parallelism approaches 186
Summary 189
For more information 189

CHAPTER 11 Visual Search Optimization 191
Image-matching application 192
Image acquisition and processing 192
Scale-space extrema detection 194
Keypoint localization 195
Orientation assignment 195
Keypoint descriptor 195
Keypoint matching 195
Applications 196
Hospitality and retail industry 197
Social interactions 197
Surveillance 197
A study of parallelism in the visual search application 197
Database (DB) level parallelism 198
Flann library parallelism 199
Experimental evaluation 202
Setup 202
Database threads scaling 202
Flann threads scaling 204
KD-tree scaling with dbthreads 204
Summary 208
For more information 208

CHAPTER 12 Radio Frequency Ray Tracing 211
Background 212
StingRay system architecture 213

Optimization examples 217
Parallel RF simulation with OpenMP 218
Parallel RF visualization with ispc 224
Summary 226
Acknowledgments 227
For more information 227

CHAPTER 13 Exploring Use of the Reserved Core 229
The Uintah computational framework 229
Radiation modeling with the UCF 230
Cross-compiling the UCF 230
Toward demystifying the reserved core 234
Exploring thread affinity patterns 234
Thread placement with PThreads 235
Implementing scatter affinity with PThreads 236
Experimental discussion 237
Machine configuration 237
Simulation configuration 237
Coprocessor-side results 238
Host-side results 240
Further analysis 240
Summary 241
Acknowledgments 242
For more information 242

CHAPTER 14 High Performance Python Offloading 243
Background 243
The pyMIC offload module 244
Design of pyMIC 244
The high-level interface 245
The low-level interface 248
Example: singular value decomposition 249
GPAW 252
Overview 252
DFT algorithm 253
Offloading 255
PyFR 260
Overview 260
Runtime code generation 260
Offloading 261

Performance ... 262
Performance of pyMIC ... 264
GPAW ... 265
PyFR ... 267
Summary ... 268
Acknowledgments ... 268
For more information ... 268

CHAPTER 15 Fast Matrix Computations on Heterogeneous Streams ... 271
The challenge of heterogeneous computing ... 271
Matrix multiply ... 272
Basic matrix multiply ... 273
Tiling for task concurrency ... 273
Heterogeneous streaming: concurrency among computing domains ... 275
Pipelining within a stream ... 275
Stream concurrency within a computing domain ... 276
Trade-offs in pipelining, tiling, and offload ... 277
Trade-offs in degree of tiling and number of streams ... 280
Tiled hStreams algorithm ... 282
The hStreams library and framework ... 284
Features ... 286
How it works ... 288
Related work ... 289
Cholesky factorization ... 290
Performance ... 294
LU factorization ... 296
Continuing work on hStreams ... 297
Acknowledgments ... 297
Recap ... 297
Summary ... 298
For more information ... 299
Tiled hStreams matrix multiplier example source ... 301

CHAPTER 16 MPI-3 Shared Memory Programming Introduction ... 305
Motivation ... 305
MPI's interprocess shared memory extension ... 306
When to use MPI interprocess shared memory ... 306
1-D ring: from MPI messaging to shared memory ... 307
Modifying MPPTEST halo exchange to include MPI SHM ... 311
Evaluation environment and results ... 313

Summary ... 318
For more information ... 319

CHAPTER 17 Coarse-Grained OpenMP for Scalable Hybrid Parallelism ... 321
Coarse-grained versus fine-grained parallelism ... 321
Flesh on the bones: A FORTRAN "stencil-test" example ... 323
Fine-grained OpenMP code ... 323
Partial coarse-grained OpenMP code ... 324
Fully coarse-grained OpenMP code ... 325
Performance results with the stencil code ... 329
Parallelism in numerical weather prediction models ... 332
Summary ... 333
For more information ... 334

CHAPTER 18 Exploiting Multilevel Parallelism in Quantum Simulations ... 335
Science: better approximate solutions ... 335
About the reference application ... 337
Parallelism in ES applications ... 338
Multicore and many-core architectures for quantum simulations ... 341
OpenMP 4.0 affinity and hot teams of intel OpenMP runtime ... 341
Hot teams motivation ... 342
Setting up experiments ... 343
MPI versus OpenMP ... 344
DGEMM experiments ... 346
FFT experiments ... 347
User code experiments ... 347
Summary: Try multilevel parallelism in your applications ... 352
For more information ... 353

CHAPTER 19 OpenCL: There and Back Again ... 355
The GPU-HEOM application ... 355
The Hexciton kernel ... 356
Building expectations ... 356
Optimizing the OpenCL Hexciton kernel ... 357
OpenCL in a nutshell ... 357
The Hexciton kernel OpenCL implementation ... 358
Vectorization in OpenCL ... 360
Memory layout optimization ... 360
Compile-time constants ... 363
Prefetching ... 363

OpenCL performance results ... 364
Performance portability in OpenCL ... 365
Nvidia GPGPU-specific optimizations ... 367
OpenCL performance portability results ... 370
Porting the OpenCL kernel to OpenMP 4.0 ... 371
OpenMP 4.0 vs. OpenCL ... 371
Substituting the OpenCL runtime ... 371
C++ SIMD libraries ... 372
Further optimizing the OpenMP 4.0 kernel ... 373
OpenMP benchmark results ... 374
Summary ... 376
Acknowledgments ... 377
For more information ... 377

CHAPTER 20 OpenMP Versus OpenCL: Difference in Performance? ... 379
Five benchmarks ... 379
Experimental setup and time measurements ... 380
HotSpot benchmark optimization ... 381
Avoiding an array copy ... 383
Applying blocking and reducing the number of instructions ... 384
Changing divisions to reciprocals ... 386
Vectorization of the inner block code ... 386
Final touch: Large pages and affinity ... 392
HotSpot optimization conclusions ... 392
Optimization steps for the other four benchmarks ... 394
LUD benchmark ... 394
CFD benchmark ... 395
NW benchmark ... 397
BFS benchmark ... 397
Summary ... 397
For more information ... 399

CHAPTER 21 Prefetch Tuning Optimizations ... 401
The importance of prefetching for performance ... 401
Prefetching on Intel Xeon Phi coprocessors ... 402
Software prefetching ... 403
Compiler intrinsics for VPREFETCH instructions ... 403
Hardware prefetching ... 403
Throughput applications ... 404
Stream Triad ... 404

Smith-Waterman ... 405
SHOC MD ... 405
Tuning prefetching ... 405
Prefetch distance tuning on the coprocessor ... 407
Results—Prefetch tuning examples on a coprocessor ... 407
Prefetching metrics for Stream Triad ... 407
Useful coprocessor hardware event counters for more in-depth analysis ... 409
Compiler prefetch distance tuning for Stream Triad ... 410
Compiler prefetch distance tuning for Smith-Waterman ... 411
Using intrinsic prefetches for hard-to-predict access patterns in SHOC MD ... 412
Results—Tuning hardware prefetching on a processor ... 416
Tuning hardware prefetching for stream on a processor ... 416
Tuning hardware prefetching for Smith-Waterman on a processor ... 416
Tuning hardware prefetching for SHOC MD on a processor ... 417
Summary ... 418
Acknowledgments ... 418
For more information ... 418

CHAPTER 22 SIMD Functions Via OpenMP ... 421
SIMD vectorization overview ... 422
Directive guided vectorization ... 423
Loop vectorization ... 423
SIMD-enabled functions ... 427
Targeting specific architectures ... 429
Optimizing for an architecture with compiler options and the processor(…) clause ... 429
Supporting multiple processor types ... 432
Vector functions in C++ ... 433
uniform(this) clause ... 434
Vector functions in Fortran ... 438
Performance results ... 438
Summary ... 438
For more information ... 440

CHAPTER 23 Vectorization Advice ... 441
The importance of vectorization ... 441
About DL_MESO LBE ... 442
Intel vectorization advisor and the underlying technology ... 443
A life cycle for experimentation ... 444

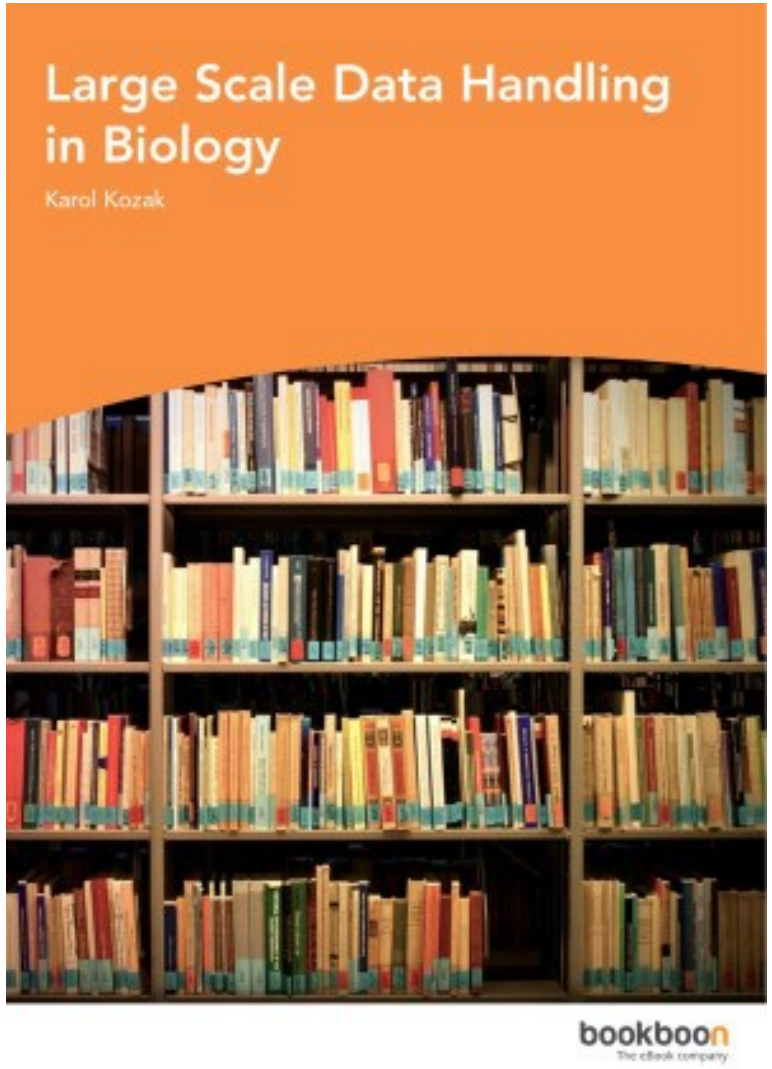
Analyzing the Lattice Boltzmann code 446
Optimizing the compute equilibrium loop (lbpSUB:744) 446
Analysis of the calculate mass density loop (lbpGET:42) 450
Balancing overheads and optimization trade-offs 453
Optimizing the move particles to neighboring grid loop (lbpSUB:1247) 453
Exploring possible vectorization strategies. 455
The results. 461
Summary 461
For more information 462

CHAPTER 24 Portable Explicit Vectorization Intrinsics 463
Related work 464
Why vectorization? 464
Portable vectorization with OpenVec 466
The vector type 467
Memory allocation and alignment 468
Built-in functions 469
if else vectorization 472
To mask or not to mask? 474
Handling vector tails 475
Math reductions 475
Compiling OpenVec code 477
Real-world example 478
Performance results 480
Developing toward the future 482
Summary 484
For more information 485

CHAPTER 25 Power Analysis for Applications and Data Centers 487
Introduction to measuring and saving power 487
Motivation to act: Importance of power optimization 487
Processor features: Modern power management features 488
Thread mapping matters: OpenMP affinity 488
Application: Power measurement and analysis 489
Basic technique 489
Methodology 489
Data collection 492
Analysis 493
Data center: Interpretation via waterfall power data charts 498
NWPerf 499

Waterfall charts generated with CView from NWPerf data 500
Installing and configuring NWPerf and CView 500
Interpreting the waterfall charts 503
Summary 509
For more information 510

Author Index 511
Subject Index 513



- Large Scale Data Handling in Biology
- *Karol Kozak*