深入理解大数据-大数据处理与编程实践

Ch.2. MapReduce简介

南京大学计算机科学与技术系

主讲人：黄宜华，顾荣

鸣谢：本课程得到Google（北京）与Intel公司
中国大学合作部精品课程计划资助

Ch.2. MapReduce简介

1.对付大数据处理-分而治之

2.构建抽象模型-Map和Reduce

3.上升到构架-自动并行化并隐藏低层细节

4.MapReduce的主要设计思想和特征

大规模数据处理时，MapReduce在三个层面上的基本构思

如何对付大数据处理：分而治之

对相互间不具有计算依赖关系的大数据，实现并行最自然的办法就是采取分而治之的策略

上升到抽象模型：Mapper与Reducer

MPI等并行计算方法缺少高层并行编程模型，为了克服这一缺陷，MapReduce借鉴了Lisp函数式语言中的思想，用Map和Reduce两个函数提供了高层的并行编程抽象模型

上升到构架：统一构架，为程序员隐藏系统层细节

MPI等并行计算方法缺少统一的计算框架支持，程序员需要考虑数据存储、划分、分发、结果收集、错误恢复等诸多细节；为此，MapReduce设计并提供了统一的计算框架，为程序员隐藏了绝大多数系统层面的处理细节

1. 如何对付大数据处理：分而治之

什么样的计算任务可进行并行化计算？

并行计算的第一个重要问题是如何划分计算任务或者计算数据以便对划分的子任务或数据块同时进行计算。

但一些计算问题恰恰无法进行这样的划分！

English Proverb:

Nine women cannot have a baby in one month!

例如：Fibonacci函数：　$F_{k+2} = F_k + F_{k+1}$

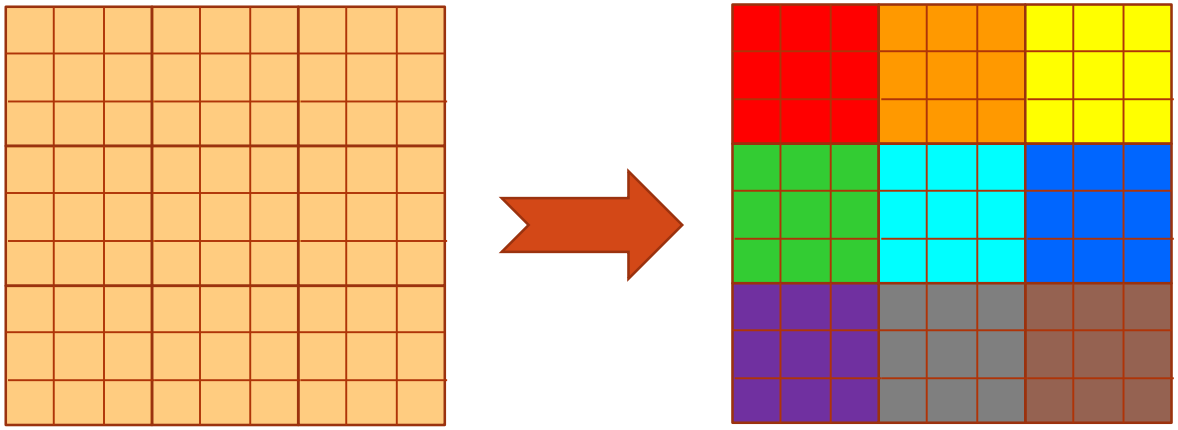
前后数据项之间存在很强的依赖关系！只能串行计算！

结论：不可分拆的计算任务或相互间有依赖关系的数据无法进行并行计算！

大数据的并行化计算

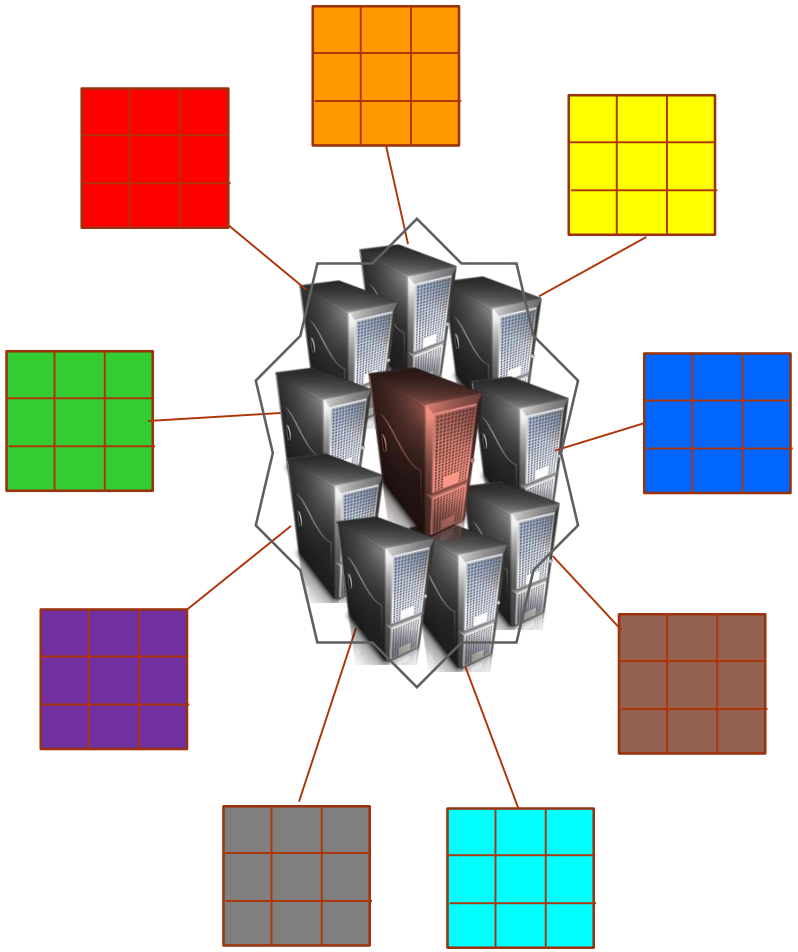
一个大数据若可以分为具有同样计算过程的数据块，并且这些数据块之间不存在数据依赖关系，则提高处理速度的最好办法就是并行计算

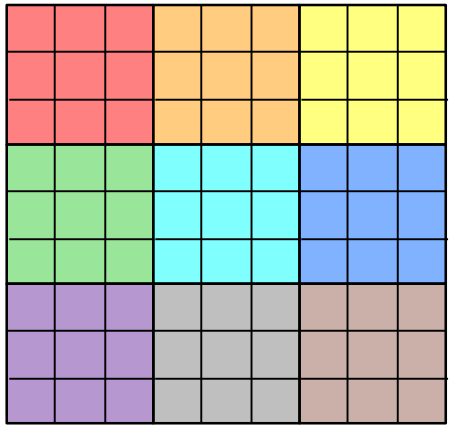
例如：假设有一个巨大的2维数据需要处理(比如求每个元素的开立方)，其中对每个元素的处理是相同的，并且数据元素间不存在数据依赖关系，可以考虑不同的划分方法将其划分为子数组，由一组处理器并行处理



如何对付大数据处理：分而治之

大数据的并行化计算

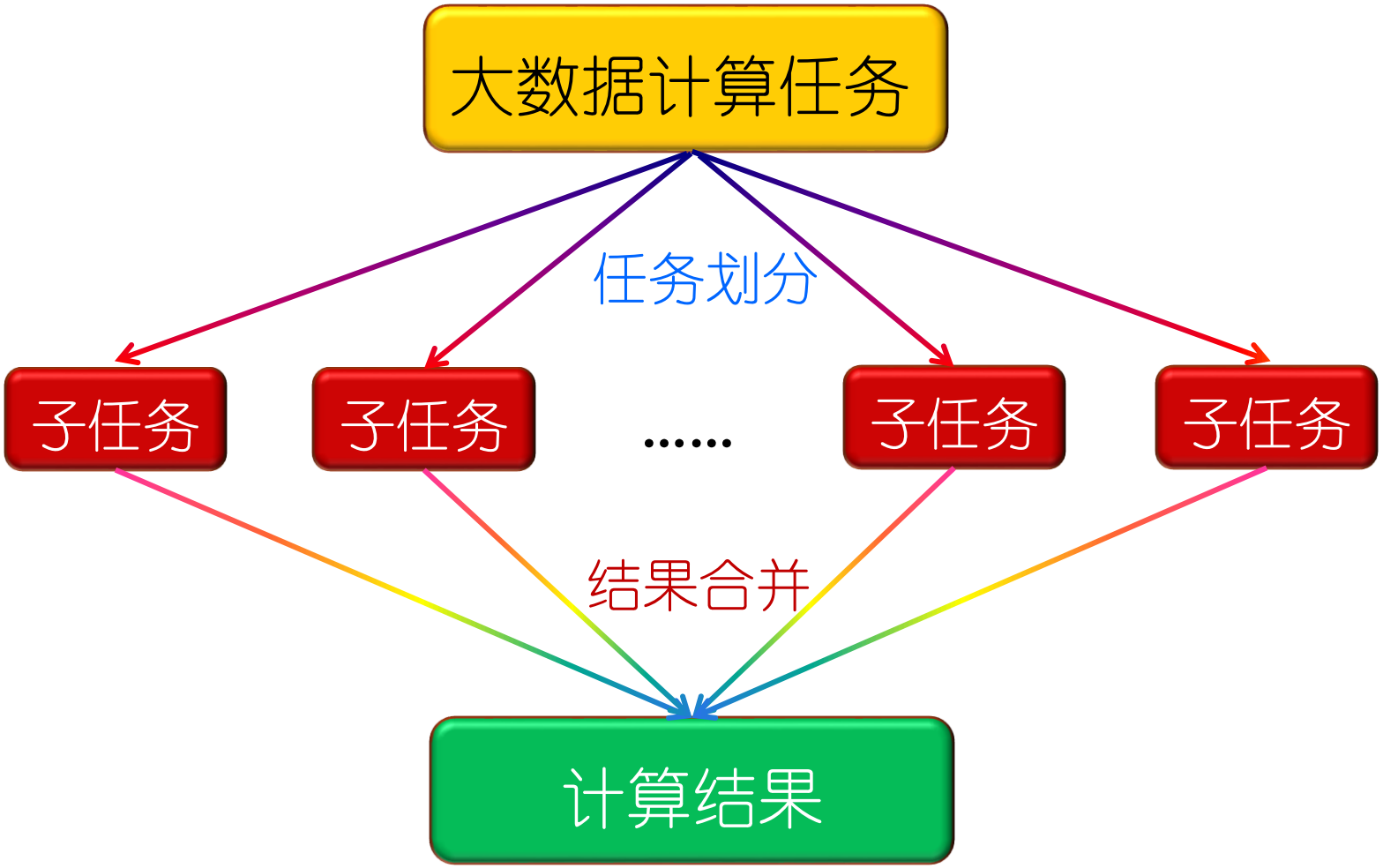




Master：负责划分和分配任务

Workder：负责数据块计算

大数据任务划分和并行计算模型



2. 构建抽象模型：Map与Reduce

借鉴函数式设计语言Lisp的设计思想

- 函数式程序设计(functional programming)语言Lisp是一种列表处理语言(List processing)，是一种应用于人工智能处理的符号式语言，由MIT的人工智能专家、图灵奖获得者John McCarthy于1958年设计发明。
- Lisp定义了可对列表元素进行整体处理的各种操作，如：

  如：(add #(1 2 3 4) #(4 3 2 1)) 将产生结果： #(5 5 5 5)
- Lisp中也提供了类似于Map和Reduce的操作

  如: (map 'vector #+ #(1 2 3 4 5) #(10 11 12 13 14))

  通过定义加法map运算将2个向量相加产生结果#(11 13 15 17 19)

  (reduce #'+ #(11 13 15 17 19)) 通过加法归并产生累加结果75

Map: 对一组数据元素进行某种重复式的处理
Reduce: 对Map的中间结果进行某种进一步的结果整理

MPI中的数据规约操作Reduce

MPI规约操作编程示例—计算积分(参见Ch. 1)

```
for(i=myid;i<N;i=i+numprocs)  /* 根据节点数目将N个矩形分为图示的多个颜色组 */
 { /* 每个节点计算一个颜色组的矩形面积并累加*/
   x = a + i*dx +dx/2; /*  以每个矩形的中心点x值计算矩形高度  */
   local +=x*x*dx;  /* 矩形面积 = 高度x宽度=y*dx  */
 }
 MPI_Reduce(&local,&inte,1,MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
 if(myid==0)                    /* 规约所有节点上的累加和并送到主节点0 */
 { /* 主节点打印累加和*/
   printf("The integal of x*x in region [%d,%d] =%16.15f\n", a, b, inte);
 }
 MPI_Finalize();
}
```

```
The integal of x*x in region[0, 10] = 333.33345
```

MPI中的数据规约操作Reduce

将一组进程的数据按照指定的操作方式规约到一起并传送给一个进程

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)

其中规约操作op可设为下表定义的操作之一：

MPI_MAX	求最大值	MPI_MIN	求最小值
MPI_SUM	求和	MPI_PROD	求积
MPI_LAND	逻辑与	MPI_BAND	按位与
MPI_LOR	逻辑或	MPI_BOR	按位或
MPI_LXOR	逻辑异或	MPI_BXOR	按位异或
MPI_MAXLOC	最大值和位置	MPI_MINLOC	最小值和位置

不足：仅能处理以上规定的规约操作，
不能实现灵活复杂的规约操作！

关系数据库中的聚合函数

对一个查询操作的结果列表中的字段表达式进行聚合操作

select Order_ID, Payment=**SUM**(Price*Quantity) groupby Order_ID

Order_ID	Item	Price	Quantity
1	电脑	5000	2
1	打印机	4000	1
1	硬盘	800	3
2	电脑	6000	1
2	硬盘	600	2

查询结果：

Orde_ID	Payment	
1	16400	(5000*2+4000*1+800*3)
2	7200	(6000*1+600*2)

Sum()	计算表达式所有值之和
Avg()	计算表达式的平均值
Count(*)	计算某字段中所有值的个数
Min()	计算表达式的最小值
Max()	计算表达式的最大值

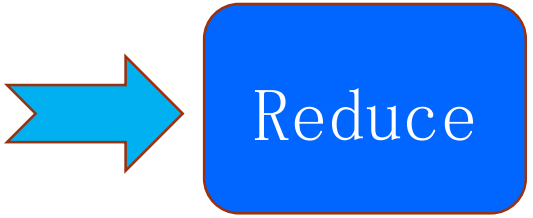
数据库中的这些聚合函数类似于对表格数据进行的Reduce操作

典型的流式大数据问题的特征

- 大量数据记录/元素进行重复处理
- 对每个数据记录/元素作感兴趣的处理、获取感兴趣的中间结果信息

- 排序和整理中间结果以利后续处理

- 收集整理中间结果
- 产生最终结果输出



关键思想：为大数据处理过程中的两个主要处理操作提供一种抽象机制

MapReduce中的Map和Reduce操作的抽象描述

MapReduce借鉴了函数式程序设计语言Lisp中的思想，定义了如下的Map和Reduce两个抽象的编程接口，由用户去编程实现:

- **map**: **(k1; v1)** → **[(k2; v2)]**

输入： 键值对(k1; v1)表示的数据

处理： 文档数据记录(如文本文件中的行，或数据表格中的行)将以“键值对”形式传入map函数；map函数将处理这些键值对，并以另一种键值对形式输出处理的一组键值对中间结果[(k2; v2)]

输出： 键值对[(k2; v2)]表示的一组中间数据

MapReduce中的Map和Reduce操作的抽象描述

- **reduce**: (k2; [v2]) → [(k3; v3)]

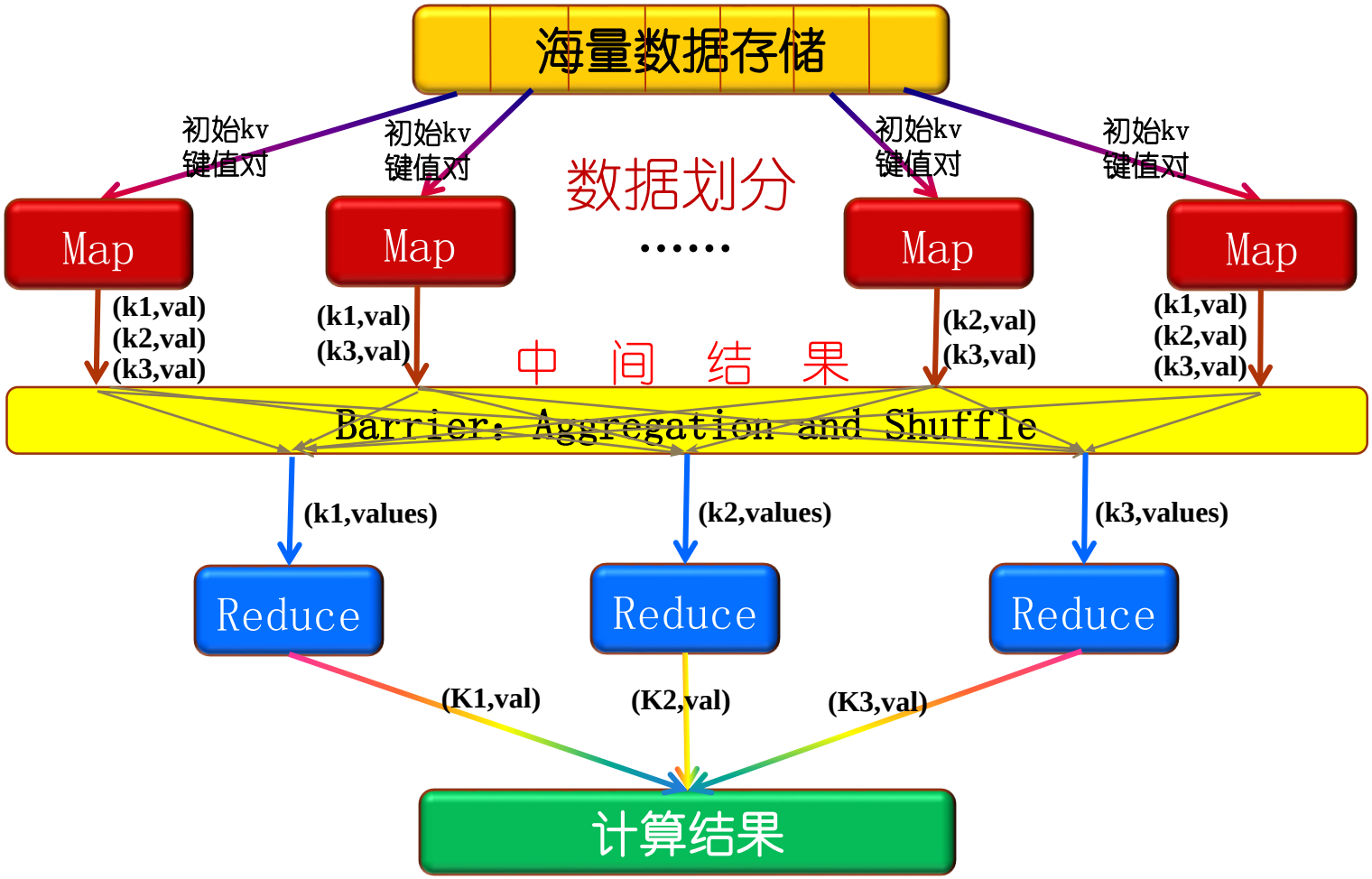
输入： 由map输出的一组键值对[(k2; v2)] 将被进行合并处理将同样主键下的不同数值合并到一个列表[v2]中，故reduce的输入为(k2; [v2])

处理：对传入的中间结果列表数据进行某种整理或进一步的处理,并产生最终的某种形式的结果输出[(k3; v3)]。

输出：最终输出结果[(k3; v3)]

Map和Reduce为程序员提供了一个清晰的操作接口抽象描述

基于Map和Reduce的并行计算模型



基于Map和Reduce的并行计算模型

- 各个map函数对所划分的数据并行处理，从不同的输入数据产生不同的中间结果输出
- 各个reduce也各自并行计算，各自负责处理不同的中间结果数据集合
- 进行reduce处理之前,必须等到所有的map函数做完，因此,在进入reduce前需要有一个同步障(barrier);这个阶段也负责对map的中间结果数据进行收集整理(aggregation & shuffle)处理,以便reduce更有效地计算最终结果
- 最终汇总所有reduce的输出结果即可获得最终结果

基于MapReduce的处理过程示例--文档词频统计：WordCount

设有4组原始文本数据：

Text 1: the weather is good　　Text 2: today is good

Text 3: good weather is good　　Text 4: today has good weather

传统的串行处理方式(Java)：

```
String[] text = new String[]
 { "the weather is good","today is good ","good weather is good ," today has good weather" } ;
HashTable ht = new HashTable();
for(i=0; i<3; ++i)
{  StringTokenizer st = new StringTokenizer(text[i]);
   while (st.hasMoreTokens())
   {  String word = st.nextToken();
      if(!ht.containsKey(word)) {  ht.put(word, new Integer(1));  }
      else { int wc = ((Integer)ht.get(word)).intValue() +1;// 计数加1
             ht.put(word, new Integer(wc));
           }
   }
}
for (Iterator itr=ht.KeySet().iterator();  itr.hasNext(); )
{ String word = (String)itr.next();
  System.out.print(word+ ": "+ (Integer)ht.get(word)+";   ");
}
```

输出：　good: 5; has: 1; is: 3; the: 1; today: 2; weather: 3

基于MapReduce的处理过程示例--文档词频统计：WordCount

MapReduce处理方式

使用4个map节点：

map节点1:

输入：(text1, “the weather is good”)

输出：(the, 1), (weather, 1), (is, 1), (good, 1)

map节点2:

输入：(text2, “today is good”)

输出：(today, 1), (is, 1), (good, 1)

map节点3:

输入：(text3, “good weather is good”)

输出：(good, 1), (weather, 1), (is, 1), (good, 1)

map节点4:

输入：(text3, “today has good weather”)

输出：(today, 1), (has, 1), (good, 1), (weather, 1)

基于MapReduce的处理过程示例--文档词频统计：WordCount

MapReduce处理方式

使用3个reduce节点：

reduce节点1:

输入：(good, 1), (good, 1), (good, 1), (good, 1), (good, 1)

输出：(good, 5)

reduce节点2:

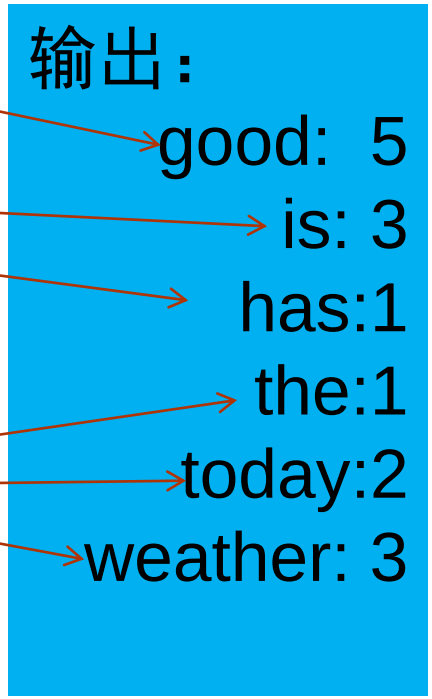
输入：(has, 1), (is,1), (is,1), (is, 1),

输出：(has, 1), (is, 3)

reduce节点3:

输入：(the, 1), (today, 1), (today, 1)
(weather, 1), (weather,1), (weather,

输出：(the, 1), (today, 2), (weather, 3)



基于MapReduce的处理过程示例--文档词频统计：WordCount

MapReduce处理方式

MapReduce伪代码(实现Map和Reduce两个函数)：

```
Class WordCountMapper
  method map(String input_key, String input_value):
    // input_key: text document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");
```

```
Class WordCountReducer
  method reduce(String output_key,
                Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v);
    Emit(output_key,  result);
```

3. 上升到构架：自动并行化并隐藏底层细节

如何提供统一的计算框架

主要需求和目标：

- 实现自动并行化计算
- 为程序员隐藏系统层细节

需要考虑的细节技术问题：

- 如何管理和存储数据？如何划分数据？
- 如何调度计算任务并分配map和reduce节点？
- 如果节点间需要共享或交换数据怎么办？
- 如何考虑数据通信和同步？
- 如何掌控节点的执行完成情况？如何收集中间和最终的结果数据？
- 节点失效如何处理？如何恢复数据？如何恢复计算任务？
- 节点扩充后如何保证原有程序仍能正常运行并保证系统性能提升？

问题：我们能把这些复杂的系统底层细节都交给系统去负责处理吗？

上升到构架：自动化并行并隐藏低层细节

如何提供统一的计算框架

答案：MapReduce之前的并行计算方法都未能做到
但MapReduce做到了！

MapReduce提供一个统一的计算框架，可完成：

- 计算任务的划分和调度
- 数据的分布存储和划分
- 处理数据与计算任务的同步
- 结果数据的收集整理(sorting, combining, partitioning,…)
- 系统通信、负载平衡、计算性能优化处理
- 处理系统节点出错检测和失效恢复

上升到构架：自动化并行并隐藏低层细节

如何提供统一的计算框架

MapReduce最大的亮点

- 通过抽象模型和计算框架把需要做什么(what need to do)与具体怎么做(how to do)分开了，为程序员提供一个抽象和高层的编程接口和框架
- 程序员仅需要关心其应用层的具体计算问题，仅需编写少量的处理应用本身计算问题的程序代码
- 如何具体完成这个并行计算任务所相关的诸多系统层细节被隐藏起来,交给计算框架去处理：从分布代码的执行，到大到数千小到单个节点集群的自动调度使用

上升到构架：自动化并行并隐藏低层细节

如何提供统一的计算框架

MapReduce提供的主要功能*

- 任务调度：提交的一个计算作业(job)将被划分为很多个计算任务(tasks), 任务调度功能主要负责为这些划分后的计算任务分配和调度计算节点(map节点或reducer节点); 同时负责监控这些节点的执行状态, 并负责map节点执行的同步控制(barrier); 也负责进行一些计算性能优化处理, 如对最慢的计算任务采用多备份执行、选最快完成者作为结果
- 数据/代码互定位：为了减少数据通信，一个基本原则是本地化数据处理(locality)，即一个计算节点尽可能处理其本地磁盘上所分布存储的数据，这实现了代码向数据的迁移；当无法进行这种本地化数据处理时，再寻找其它可用节点并将数据从网络上传送给该节点(数据向代码迁移)，但将尽可能从数据所在的本地机架上寻找可用节点以减少通信延迟

* Cite from Jimmy Lin, University of Maryland, Data-Intensive Text processing with MapReduce

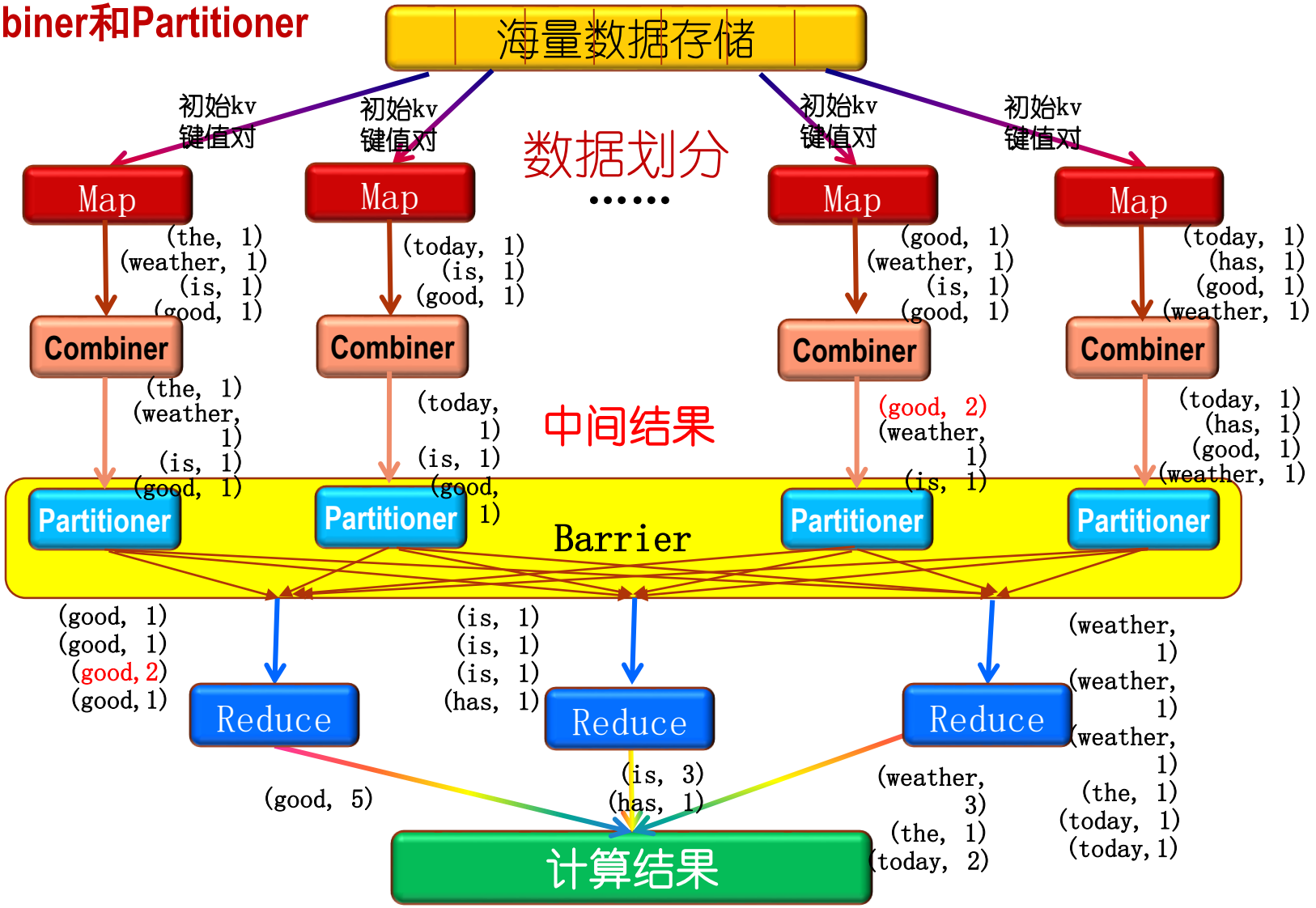
如何提供统一的计算框架

MapReduce提供的主要功能

- 出错处理：以低端商用服务器构成的大规模MapReduce计算集群中,节点硬件(主机、磁盘、内存等)出错和软件有bug是常态，因此,MapReducer需要能检测并隔离出错节点，并调度分配新的节点接管出错节点的计算任务
- 分布式数据存储与文件管理：海量数据处理需要一个良好的分布数据存储和文件管理系统支撑,该文件系统能够把海量数据分布存储在各个节点的本地磁盘上,但保持整个数据在逻辑上成为一个完整的数据文件；为了提供数据存储容错机制,该文件系统还要提供数据块的多备份存储管理能力
- Combiner和Partitioner:为了减少数据通信开销,中间结果数据进入reduce节点前需要进行合并(combine)处理,把具有同样主键的数据合并到一起避免重复传送; 一个reducer节点所处理的数据可能会来自多个map节点, 因此, map节点输出的中间结果需使用一定的策略进行适当的划分(partitioner)处理，保证相关数据发送到同一个reducer节点

基于Map和Reduce的并行计算模型

Combiner和Partitioner



4. MapReduce的主要设计思想与特点*

向“外”横向扩展，而非向“上”纵向扩展

Scale “out", not “up”

即MapReduce集群的构筑选用价格便宜、易于扩展的大量低端商用服务器，而非价格昂贵、不易扩展的高端服务器（SMP）

- 低端服务器市场与高容量Desktop PC有重叠的市场，因此，由于相互间价格的竞争、可互换的部件、和规模经济效应，使得低端服务器保持较低的价格
- 基于TPC-C在2007年底的性能评估结果，一个低端服务器平台与高端的共享存储器结构的服务器平台相比，其性价比大约要高4倍；如果把外存价格除外，低端服务器性价比大约提高12倍
- 对于大规模数据处理，由于有大量数据存储需要，显而易见，基于低端服务器的集群远比基于高端服务器的集群优越，这就是为什么MapReduce并行计算集群会基于低端服务器实现

* Cite from Jimmy Lin, University of Maryland, Data-Intensive Text processing with MapReduce

失效被认为是常态

Assume failures are common

MapReduce集群中使用大量的低端服务器(Google目前在全球共使用百万台以上的服务器节点)，因此，节点硬件失效和软件出错是常态，因而：

- 一个良好设计、具有容错性的并行计算系统不能因为节点失效而影响计算服务的质量，任何节点失效都不应当导致结果的不一致或不确定性；任何一个节点失效时，其它节点要能够无缝接管失效节点的计算任务；当失效节点恢复后应能自动无缝加入集群，而不需要管理员人工进行系统配置
- MapReduce并行计算软件框架使用了多种有效的机制，如节点自动重启技术，使集群和计算框架具有对付节点失效的健壮性，能有效处理失效节点的检测和恢复。

把处理向数据迁移

Moving processing to the data

- 传统高性能计算系统通常有很多处理器节点与一些外存储器节点相连，如用区域存储网络(SAN,Storage Area Network)连接的磁盘阵列，因此，大规模数据处理时外存文件数据I/O访问会成为一个制约系统性能的瓶颈。
- 为了减少大规模数据并行计算系统中的数据通信开销，代之以把数据传送到处理节点(数据向处理器或代码迁移)，应当考虑将处理向数据靠拢和迁移。
- MapReduce采用了数据/代码互定位的技术方法，计算节点将首先将尽量负责计算其本地存储的数据,以发挥数据本地化特点(locality),仅当节点无法处理本地数据时，再采用就近原则寻找其它可用计算节点，并把数据传送到该可用计算节点。

顺序处理数据、避免随机访问数据

Process data sequentially and avoid random access

- 大规模数据处理的特点决定了大量的数据记录不可能存放在内存、而只可能放在外存中进行处理。
- 磁盘的顺序访问和随即访问在性能上有巨大的差异

例：100亿(10^{10})个数据记录(每记录100B,共计1TB)的数据库更新1%的记录(一定是随机访问)需要1个月时间；而顺序访问并重写所有数据记录仅需1天时间！

- MapReduce设计为面向大数据集批处理的并行计算系统，所有计算都被组织成很长的流式操作，以便能利用分布在集群中大量节点上磁盘集合的高传输带宽。

为应用开发者隐藏系统层细节

Hide system-level details from the application developer

- 软件工程实践指南中，专业程序员认为之所以写程序困难，是因为程序员需要记住太多的编程细节(从变量名到复杂算法的边界情况处理)，这对大脑记忆是一个巨大的认知负担,需要高度集中注意力
- 而并行程序编写有更多困难，如需要考虑多线程中诸如同步等复杂繁琐的细节，由于并发执行中的不可预测性，程序的调试查错也十分困难；大规模数据处理时程序员需要考虑诸如数据分布存储管理、数据分发、数据通信和同步、计算结果收集等诸多细节问题
- MapReduce提供了一种抽象机制将程序员与系统层细节隔离开来，程序员仅需描述需要计算什么(what to compute), 而具体怎么去做(how to compute)就交由系统的执行框架处理，这样程序员可从系统层细节中解放出来，而致力于其应用本身计算问题的算法设计

平滑无缝的可扩展性

Seamless scalability

主要包括两层意义上的扩展性：数据扩展和系统规模扩展

- 理想的软件算法应当能随着数据规模的扩大而表现出持续的有效性，性能上的下降程度应与数据规模扩大的倍数相当
- 在集群规模上，要求算法的计算性能应能随着节点数的增加保持接近线性程度的增长
- 绝大多数现有的单机算法都达不到以上理想的要求；把中间结果数据维护在内存中的单机算法在大规模数据处理时很快失效；从单机到基于大规模集群的并行计算从根本上需要完全不同的算法设计
- 奇妙的是，MapReduce几乎能实现以上理想的扩展性特征。多项研究发现基于MapReduce的计算性能可随节点数目增长保持近似于线性的增长

MapReduce的主要设计思想与特点

Stanford大学研究小组研究基于多核构架、自行设计的轻量级MapReduce框架的各种机器学习算法，发现计算性能可随处理器核数增长保持近似于线性的增长

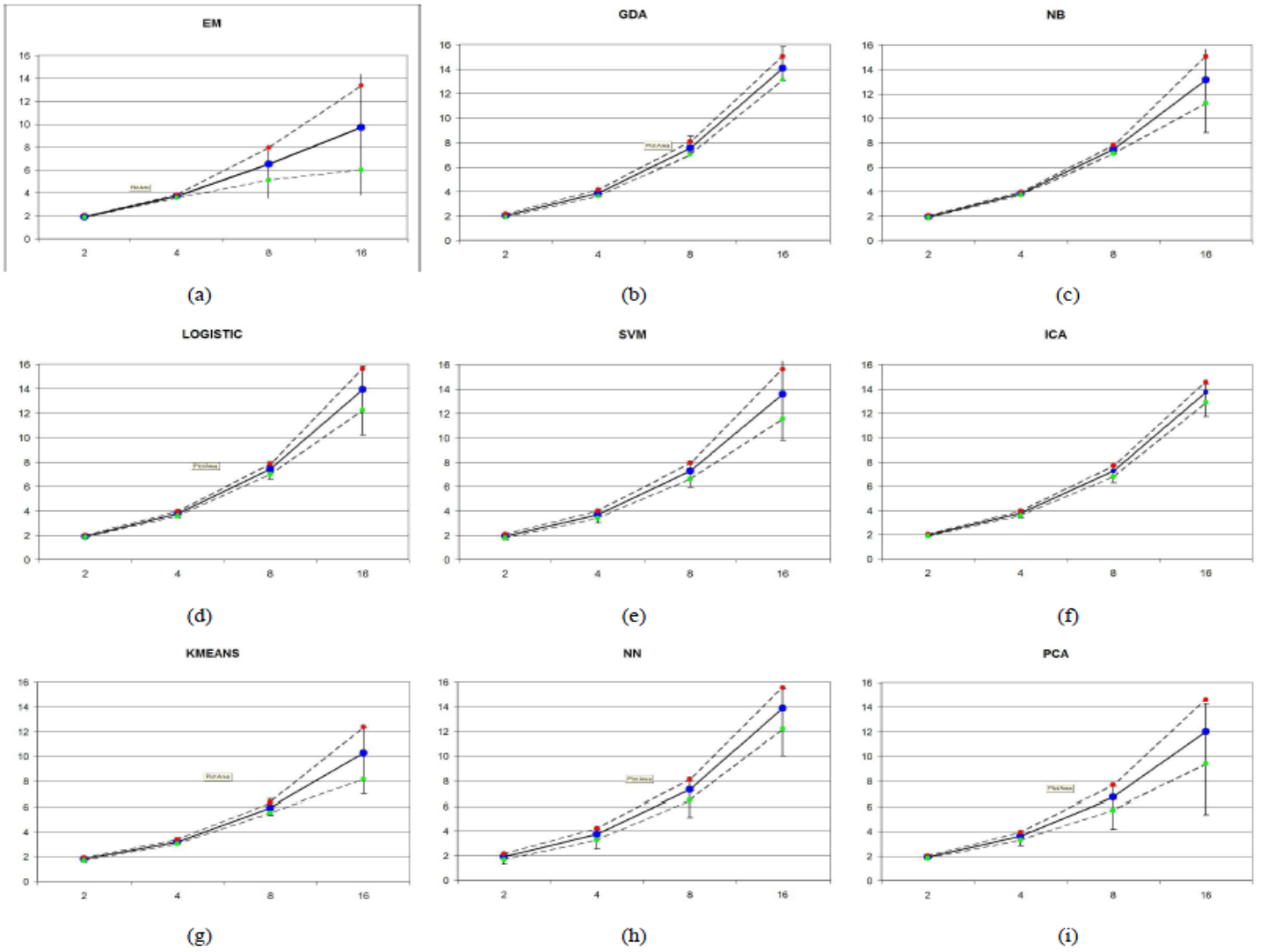
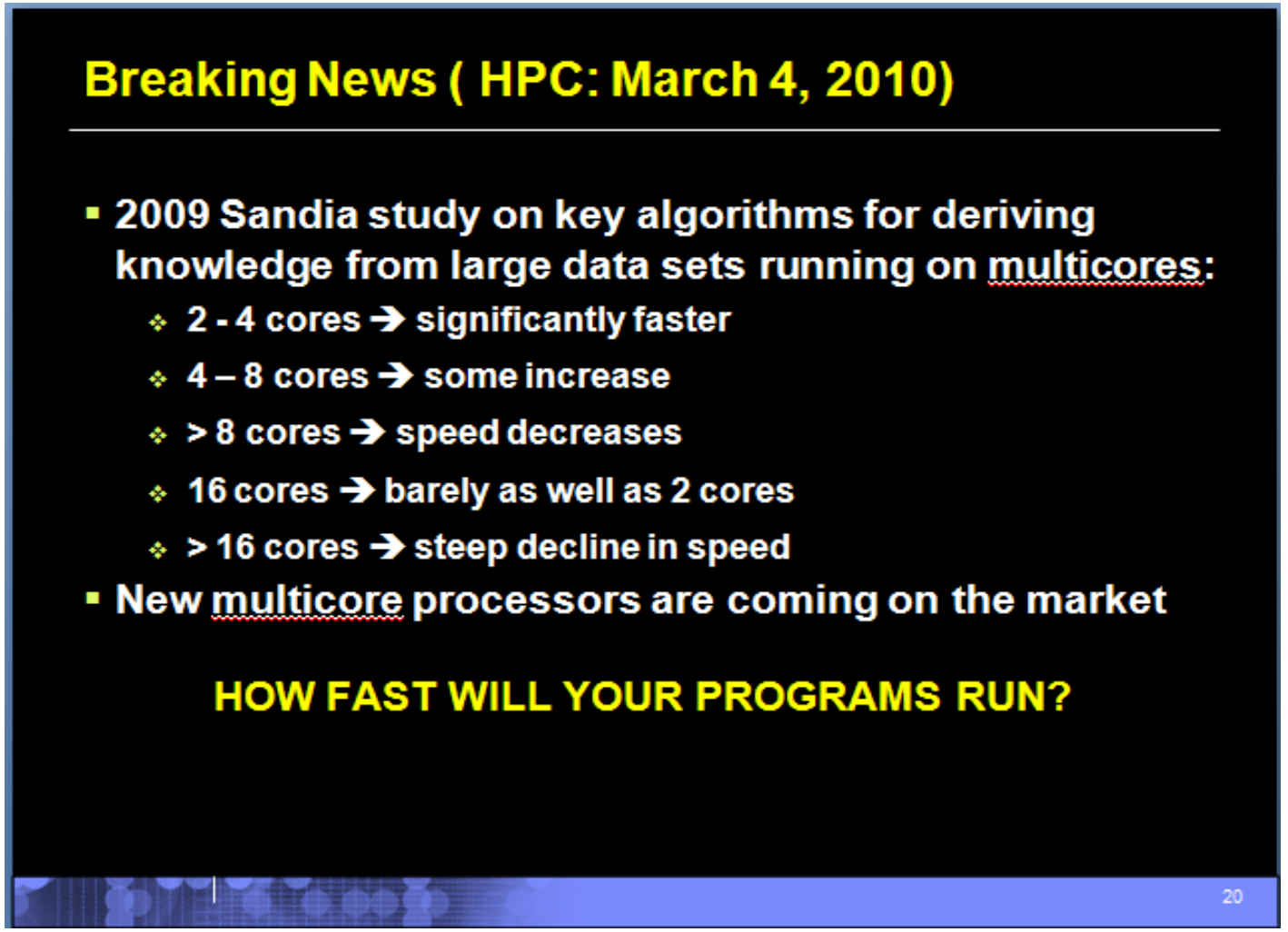


Figure 2: (a)-(i) show the speedup from 1 to 16 processors of all the algorithms over all the data sets. The Bold line is the average, error bars are the max and min speedups and the dashed lines are the variance.

Cite from Cheng-Tao Chu et.al , MapReduce for Machine Learning on Multicore, 2006

MapReduce的主要设计思想与特点

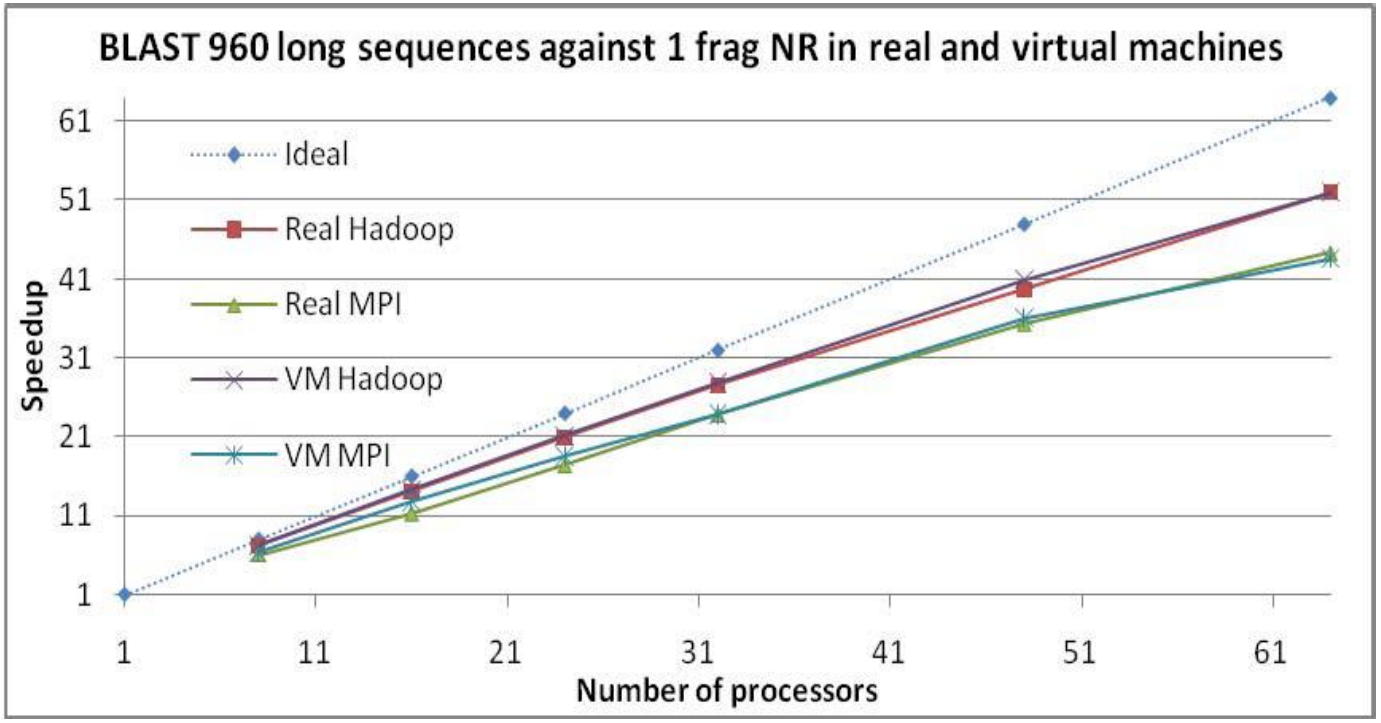
与此相比，其它基于非MapReduce构架的多核并行计算研究结果发现性能无法达到预期的增长



Cite from Fran Ellen, Keynote at Nanjing University October 18, 2010

平滑无缝的可扩展性

基于MapReduce的基因序列比对算法BLAST的研究显示,无论基于虚拟机还是非虚拟机MapReduce，随着处理器数目的增加都能实现近似于线性的性能增长



Cite from Andréa Matsunaga et.al. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. 2008

Thanks !