Scaling-up Data Mining

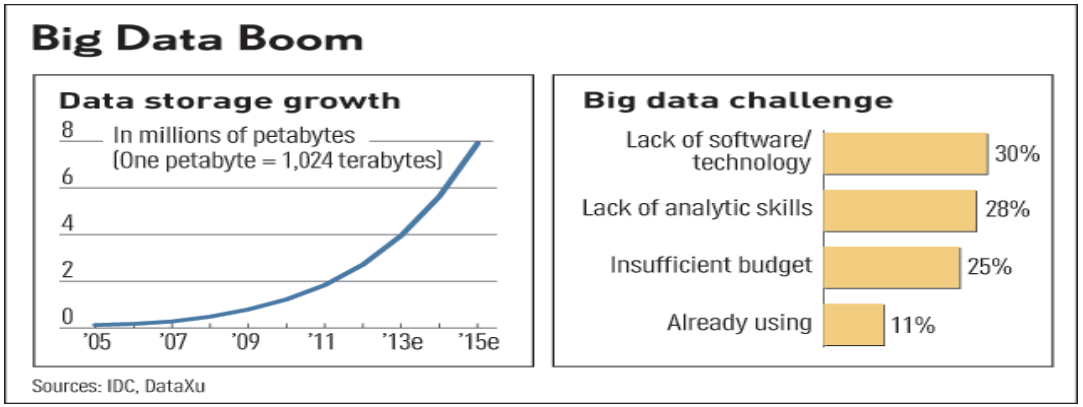
Dr. David C. Anastasiu
San José State University

Outline

- Why scale?
- How to scale?
- Shared memory computing
- pL2Knn: a shared memory example
- GPGPUs and many-core CPUs
- Message passing distributed computing
- Share-nothing distributed computing

WHY SCALE DATA MINING?

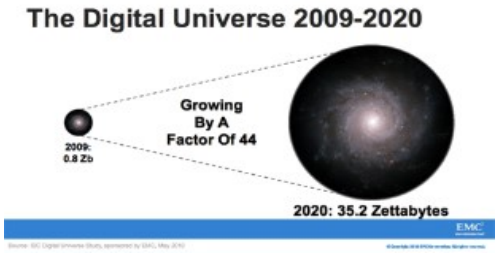
"***Big Data***" is data whose scale, diversity, and complexity require new architecture, techniques, algorithms, and analytics to manage it and extract value and hidden knowledge from it…

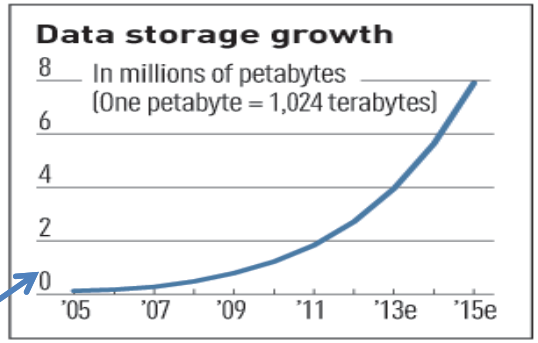


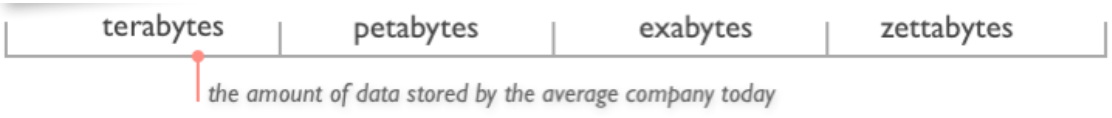
- **The Bottleneck is in technology**
  - New architecture, algorithms, techniques are needed
- **Also in technical skills**
  - Experts in using the new technology and dealing with big data

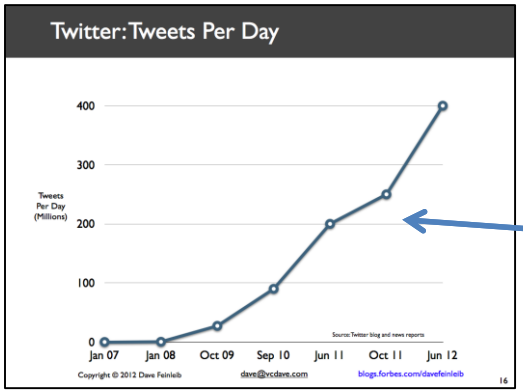
Characteristics of Big Data: 1-Scale (Volume)

- **Data Volume**
  - 44x increase from 2009 to 2020
  - From 0.8 zettabytes to 35zb
- Data volume is increasing exponentially





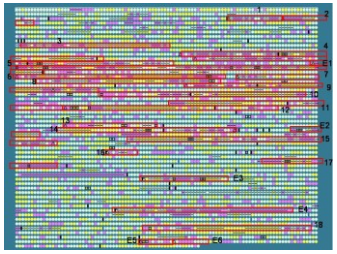


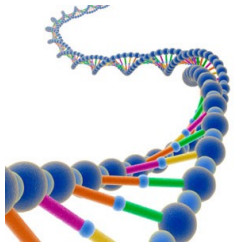


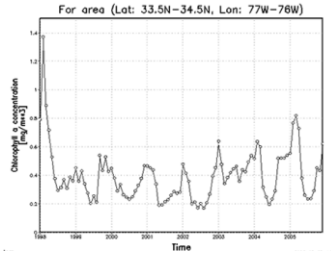
Exponential increase in collected/generated data

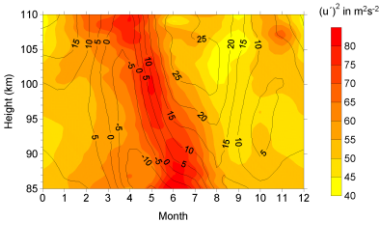
Characteristics of Big Data: 2-Complexity (Varity)

- Various formats, types, and structures
- Text, numerical, images, audio, video, sequences, time series, social media data, multi-dim arrays, etc…
- Static data vs. streaming data
- A single application can be generating/collecting many types of data







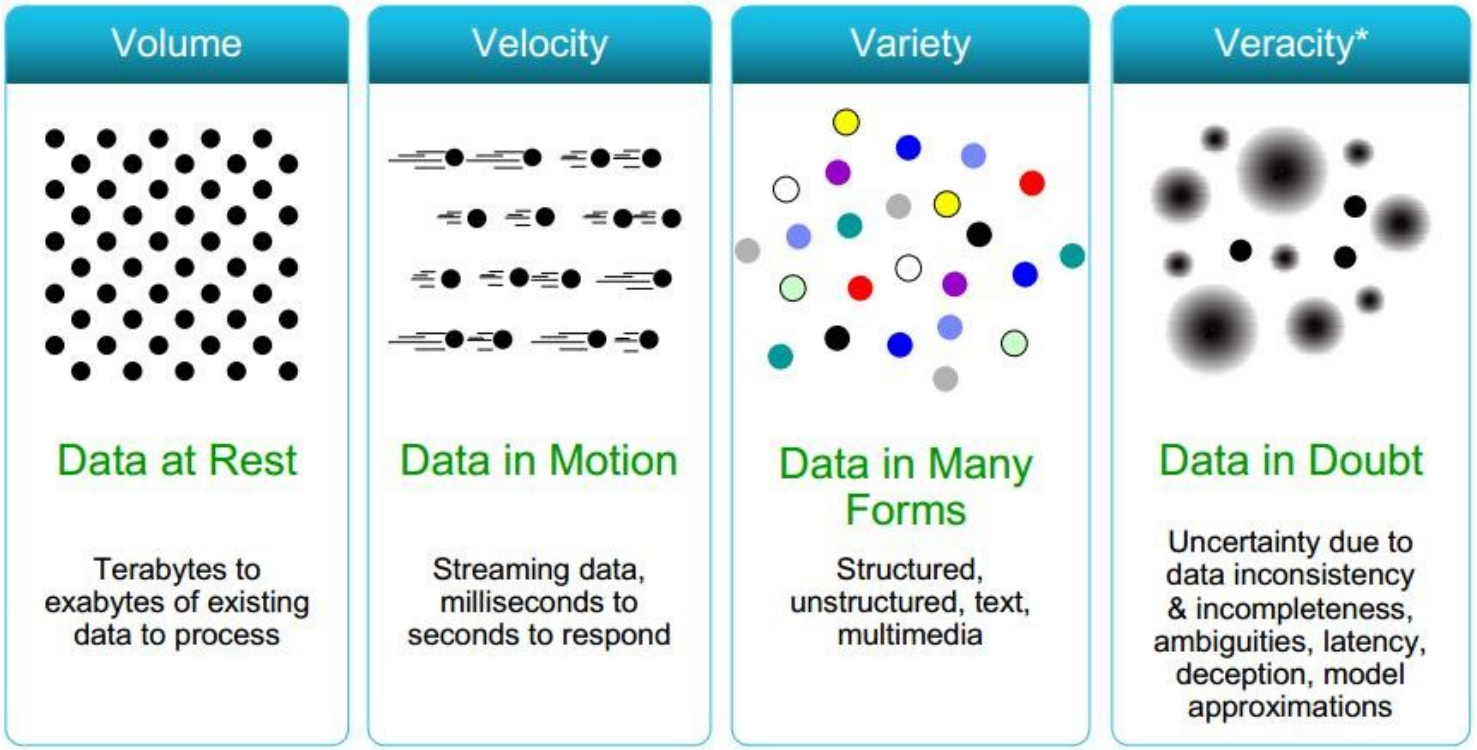


To extract knowledge➔ all these types of data need to linked together

Characteristics of Big Data: 3-Speed (Velocity)

- Data is begin generated fast and need to be processed fast
- Online Data Analytics
- Late decisions ➔ missing opportunities
- **Examples**
  - **E-Promotions:** Based on your current location, your purchase history, what you like ➔ send promotions right now for store next to you
  - **Healthcare monitoring:** sensors monitoring your activities and body ➔ any abnormal measurements require immediate reaction

Some Make it 4V's



Borrowed from Mohamed Eltabakh: web.cs.wpi.edu/~cs525/s13-MYE/lectures/1/intro.**ppt**x

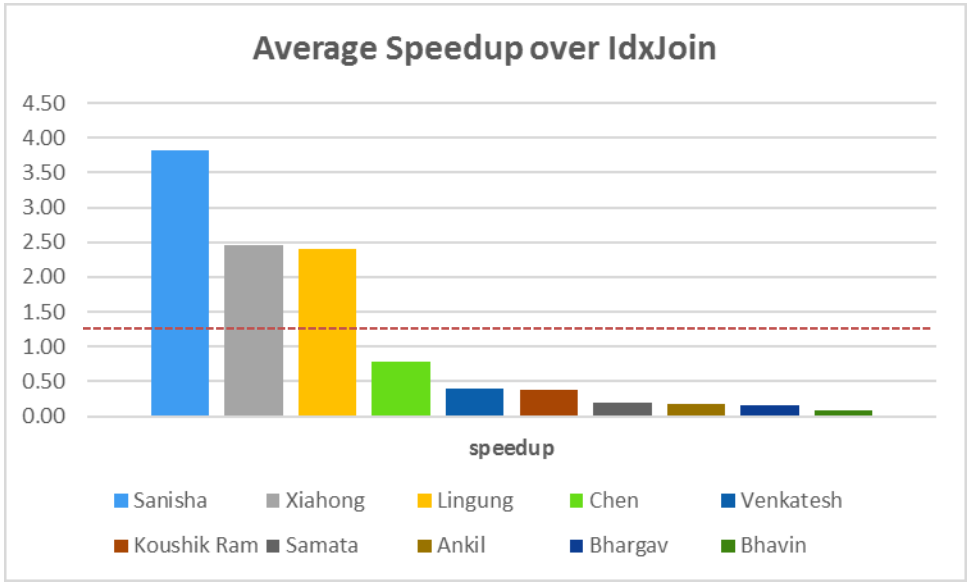
HOW TO SCALE DATA MINING?

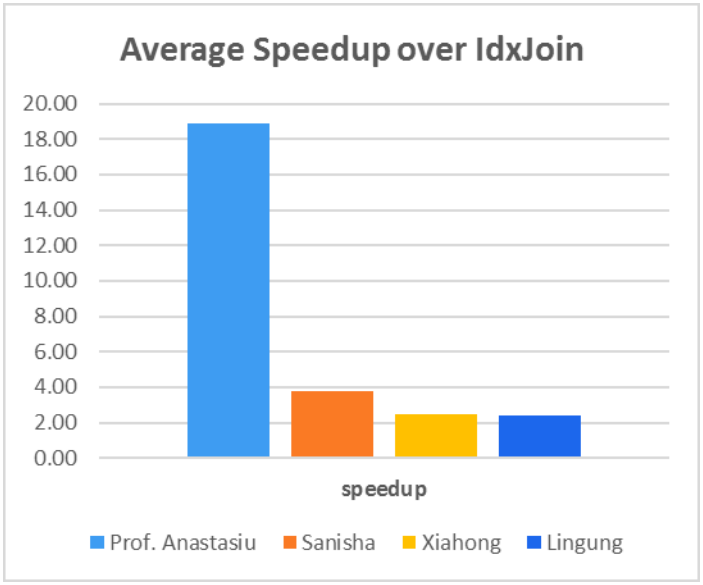
Use some of the data

- Sampling, e.g.:
  - Clustering:
    - Randomly select a subset of the users and cluster them
    - Compute cluster representatives
    - Assign all other objects to closest representative
  - Recommender systems:
    - Estimate only item factors in a MF model from a subset of the users & compute user factors on the fly.
- Approximation, e.g.,
  - Approximate nearest neighbor methods (LSH, GF)

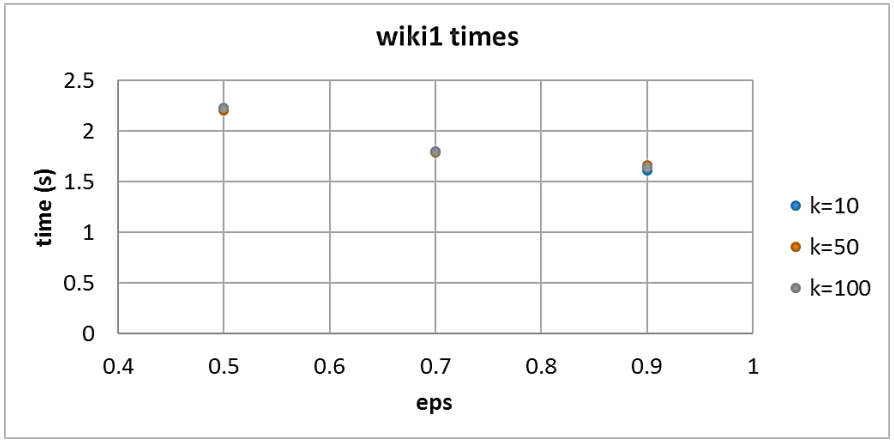
Design more efficient algorithms

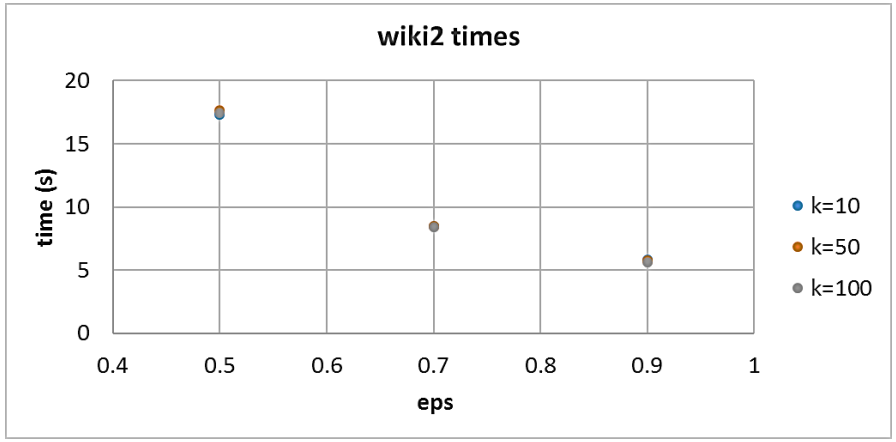
- Use appropriate libraries/languages
- Use theory to do less work/improve efficiency

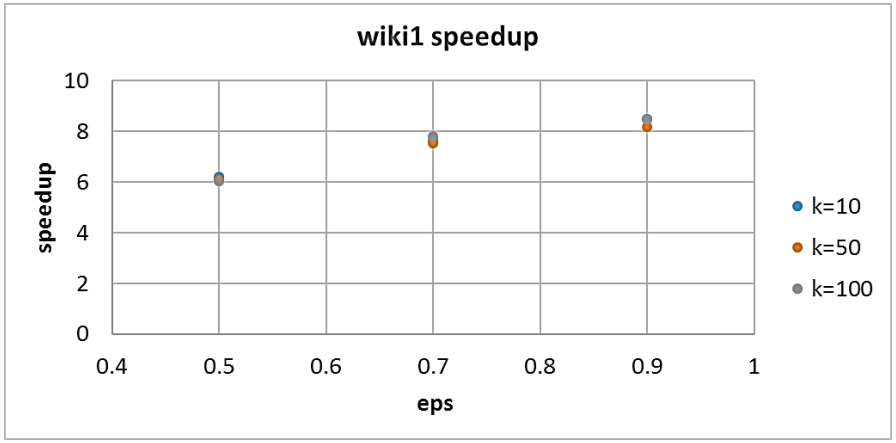


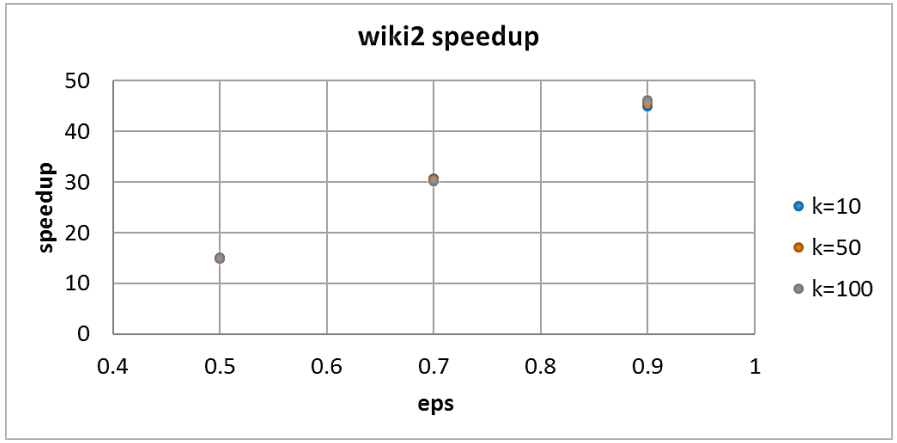


My solution


wiki1 times
time (s)
2.5
2
1.5
1
0.5
0
0.4
0.5
0.6
0.7
0.8
0.9
1
eps
k=10
k=50
k=100


wiki2 times
time (s)
20
15
10
5
0
0.4
0.5
0.6
0.7
0.8
0.9
1
eps
k=10
k=50
k=100


wiki1 speedup
speedup
10
8
6
4
2
0
0.4
0.5
0.6
0.7
0.8
0.9
1
eps
k=10
k=50
k=100


wiki2 speedup
speedup
50
40
30
20
10
0
0.4
0.5
0.6
0.7
0.8
0.9
1
eps
k=10
k=50
k=100

Use multiple processing units

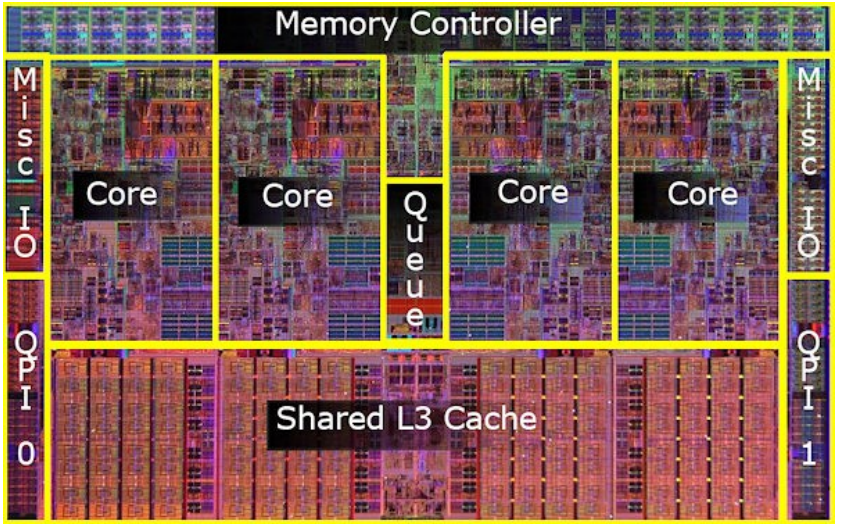
Parallel computing models:

- Shared-memory (multi-core) processing
- Accelerators
  - GPGPUs
  - many-core processors
- Distributed systems
  - Message passing
  - Share-nothing

SHARED MEMORY COMPUTING

Ubiquitous hardware

- Multi-core machines are everywhere today
  - Your smartphone
  - Your laptop
  - Your desktop
  - Your TV
  - Severs
  - Cloud computing nodes
  - Supercomputing nodes



- Define parts of program for each core to work on
  - Memory contention
  - Load balancing

OpenMP

- An API for multi-platform shared memory processing in C, C++, and Fortran
- Uses hints (pragma directives) to transform serial code into multi-threaded code
  - Parts of the program are executed concurrently on different available cores

```
int main(int argc, char **argv) {
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }
    return 0;
}
```

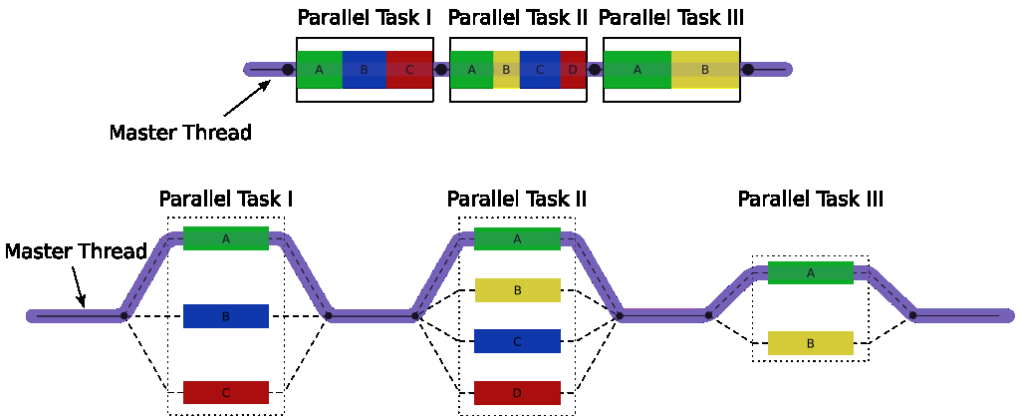


Image and example from https://en.wikipedia.org/wiki/OpenMP

A shared-memory parallel program example

PL2KNN

k-nearest neighbors

L2Knng: Filtering framework for k-NNG construction

- Problem: unknown threshold ϵ

Solution:

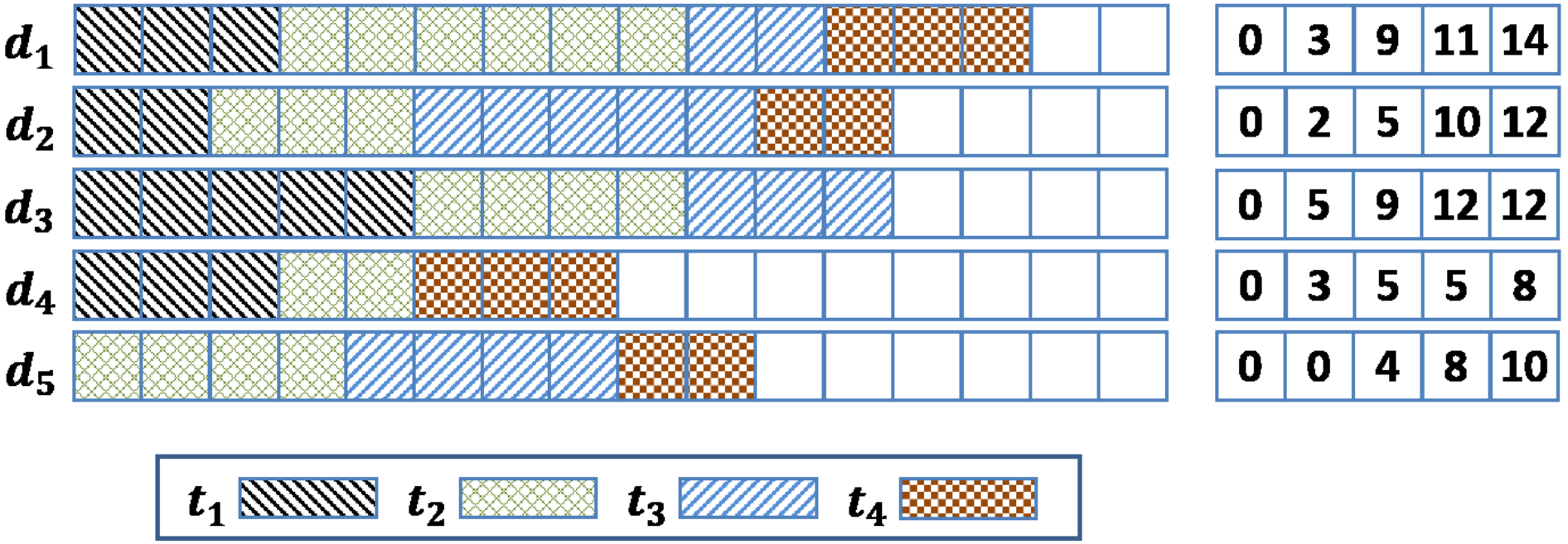
- Build an initial approximate graph $\hat{G}$
  - Provides thresholds for filtering
- Improve $\hat{G}$ until exact
  - Uses minimum similarities in $\hat{G}$ neighborhoods to filter objects that can not improve them
  - Minimum similarities change as we process objects

Cache tiling

- **pL2Knng focus**: multi-threaded solution
- 1-D variable block size decomposition
  - Inverted index block size determined dynamically
    - Maximum ζ non-zeros
    - Maximum ϵ objects
  - Static query block size
    - η objects
  - Within a query block, threads dynamically assigned micro-blocks of 20 queries
- Additional serial improvements meant to improve cache locality
  - Lead to 1.44x – 1.73x speedup

Neighborhood updates

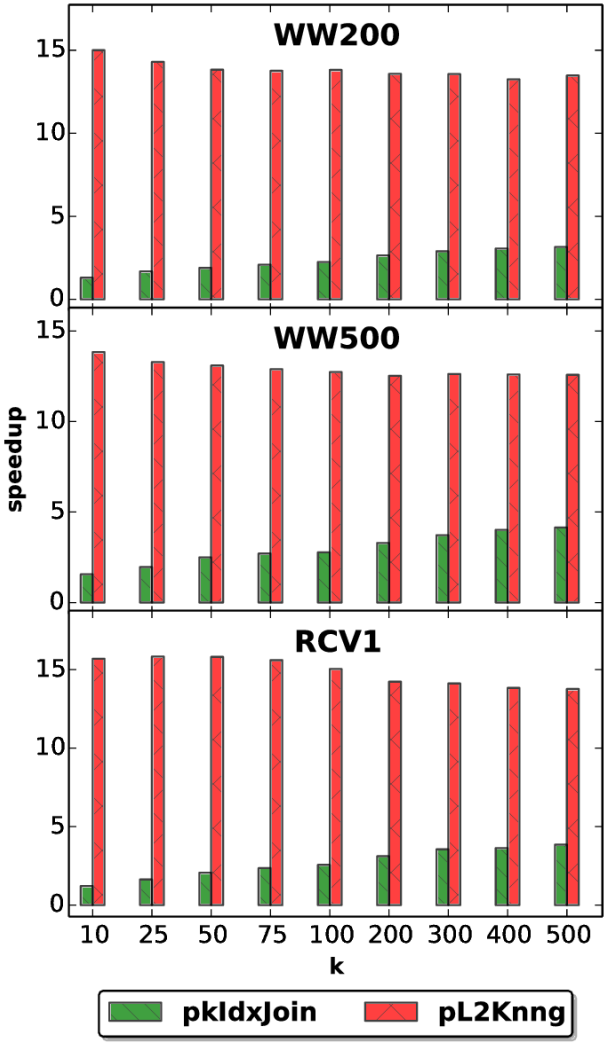
- Local neighborhood updated during search
- Candidate neighborhood updates staged for cooperative update at the end of query block



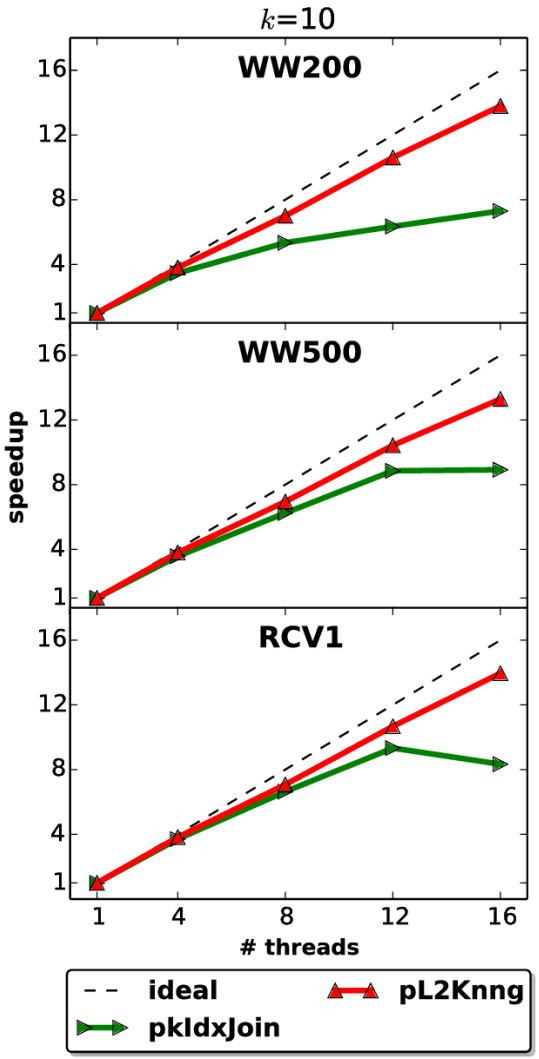
pL2Knng: Efficiency & Scaling

Baseline:
pkIdxJoin, parallel version of IdxJoin with similar cache tiling but no pruning

Dataset	n	m	nnz
RCV1	804K	46K	62M
WW200	1,018K	663K	437M
WW500	243K	661K	202M



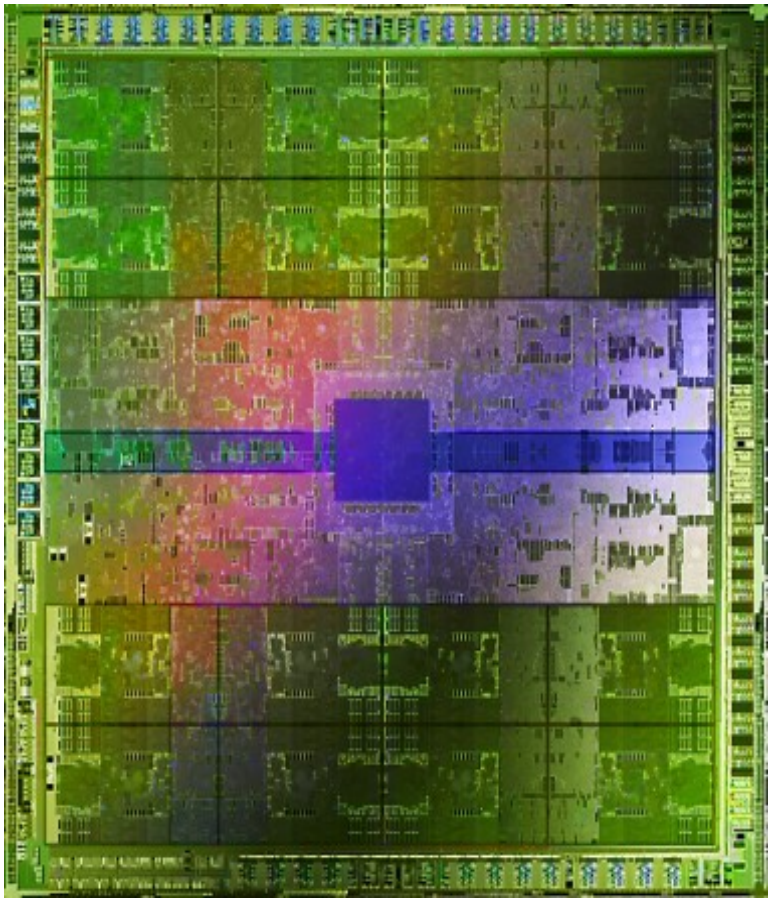
Using 16 cores, vs. best serial method (L2Knng)

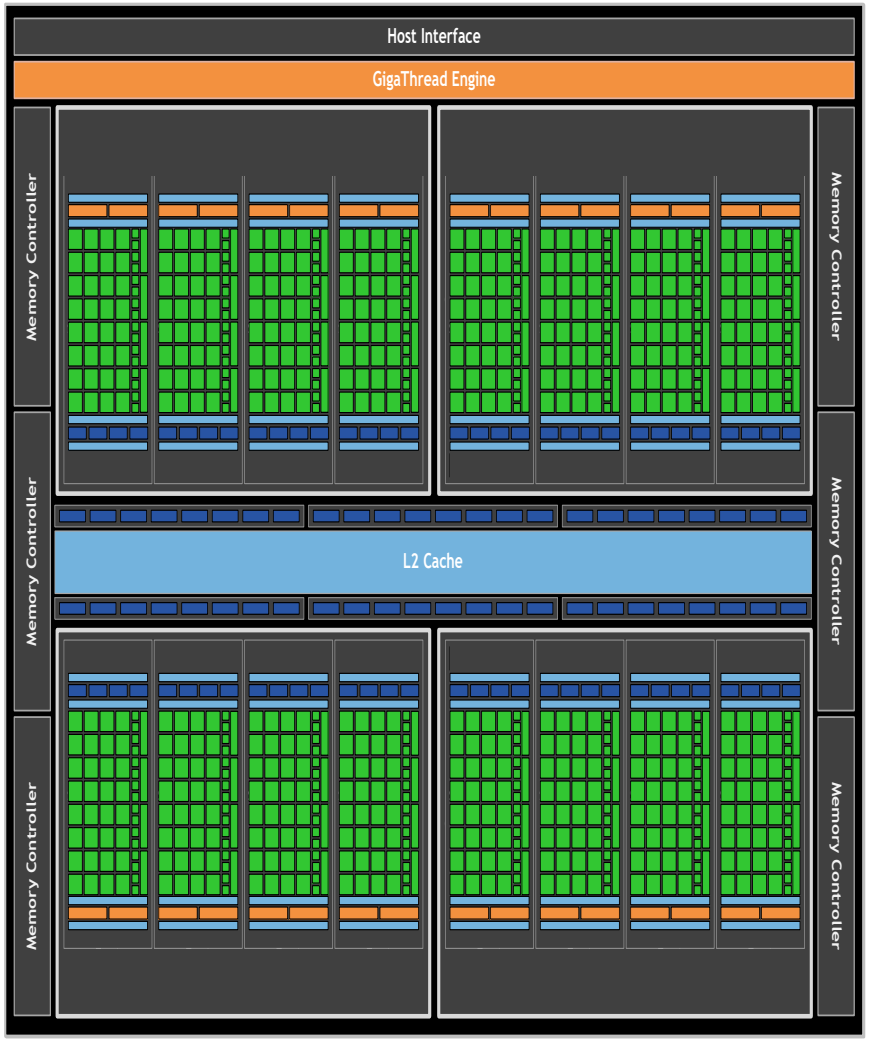


Vs. own single-threaded execution

ACCELERATORS

Fermi GF100 GPU





GeForce GTX 1080

Cores	2560
Memory	8Gb / 256-bit GDDR5X
Memory Bandwidth	320 Gb/sec
Base/Boost Clock	1607 / 1733 MHz
Power Connectors	1x 8-pin
Power	180W
Length	10.5 inches
Outputs	DP 1.4 HDMI 2.0b DL-DVI

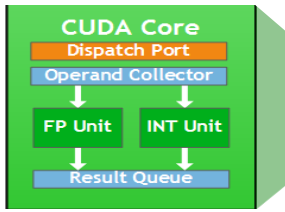
GeForce Titan X

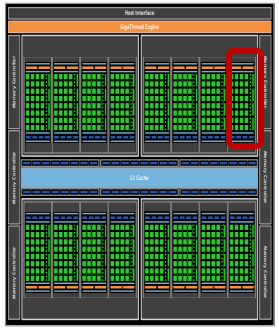
Cores	3072
Memory	12Gb / 384-bit GDDR5
Memory Bandwidth	336.5 GB/sec
Base/Boost Clock	1000 / 1075 MHz
Power Connectors	6-pin + 8-pin
Power	250W
Length	10.5 inches
Outputs	Dual Link DVI-I HDMI 2.0 3x DisplayPort 1.2

Stream Multiprocessor (SM)

- 32 cores per SM
- 64KB configurable
  L1 cache/ shared memory

- SIMT (Single Instruction Multiple Thread) execution
  - threads run in groups of 32 called **warps**
  - threads in a warp share instruction unit (IU)
  - HW automatically handles divergence along branches

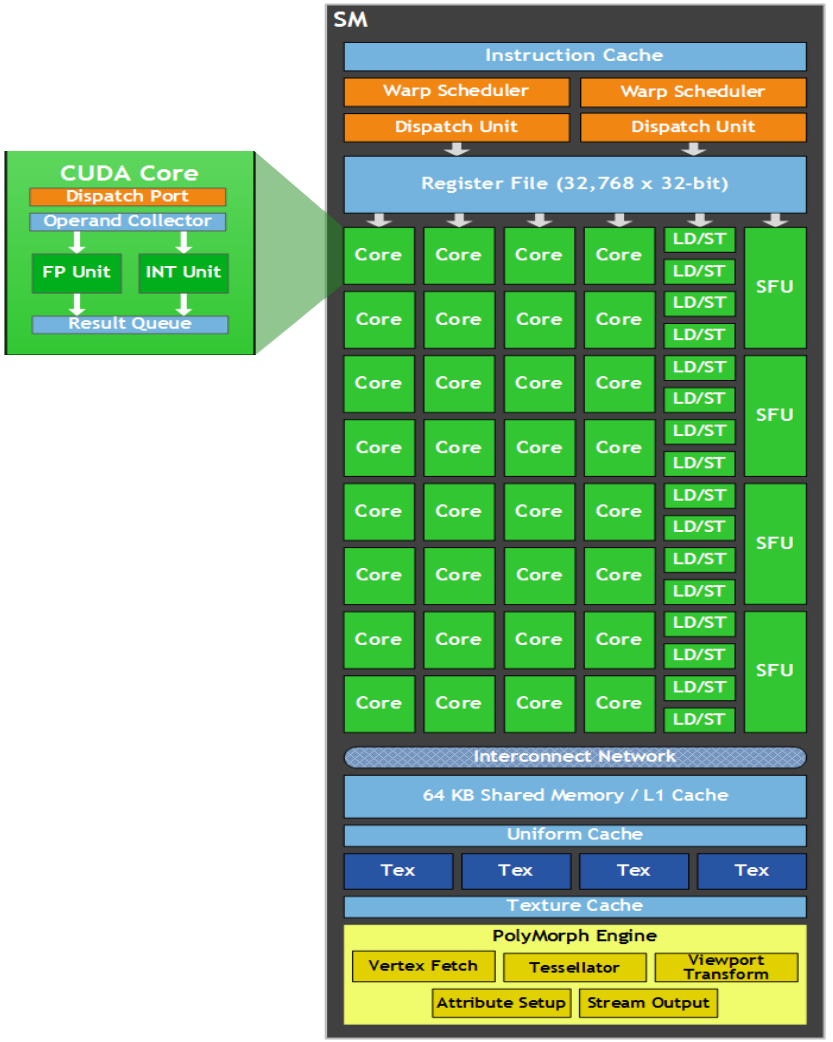




	FP32	FP64	INT	SFU	LD/ST
Ops / clk	32	16	32	4	16

Execution model

- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency
- Threads have all resources needed to run
  - any warp not waiting for something can run
  - context switching is (basically) free
- Instruction Set Architecture
  - Enables C++ : virtual functions, new/delete, try/catch
  - Unified load/store addressing
  - 64-bit addressing for large problems



CUDA Programing

- Augment C/C++ with minimalist abstractions
- Provide straightforward mapping onto hardware
  - good fit to GPU architecture
  - maps well to multi-core CPUs too
- Scale to 100s of cores & 10,000s of parallel threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization

CUDA model of parallelism

- A parallel computation is initiated by executing a kernel function.
  - It runs to completion and returns control back to the host.
- The traditional notion of processing elements is organized in a two level hierarchy:
  - A set of blocks, each containing the same number of threads.
    - You can think of asking for #blocks * #threads processing elements.
- Each block is scheduled on an SM and the threads of each block are scheduled on the cores of the SM.

C for CUDA

- Philosophy: provide minimal set of extensions necessary to expose power

- Function qualifiers:

```
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

- Variable qualifiers:

```
__constant__ float my_constant_array[32];
__shared__   float my_shared_array[32];
```

- Execution configuration:

```
dim3 grid_dim(100, 50);  // 5000 thread blocks
dim3 block_dim(4, 8, 8); // 256 threads per block
my_kernel <<< grid_dim, block_dim >>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx; // Thread index
void __syncthreads(); // Thread synchronization
```

Example: vector_addition

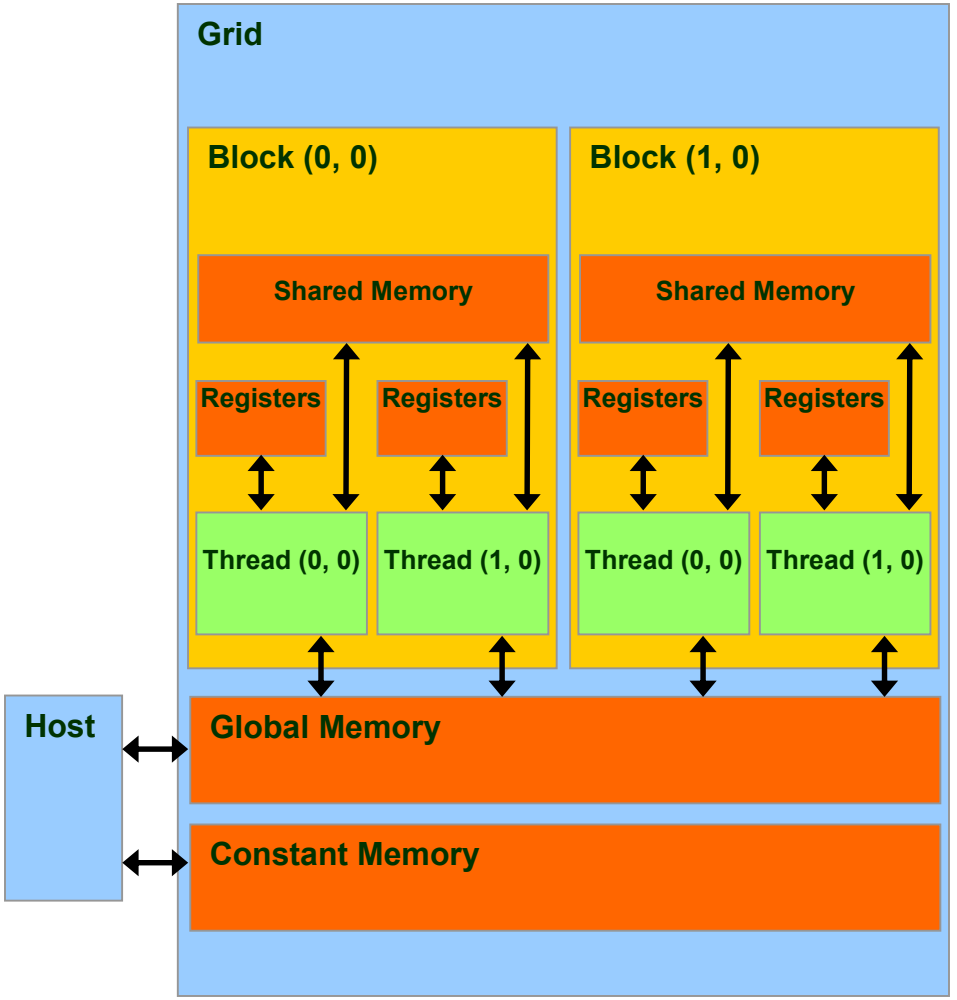
```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Initialization code
    ...
    // Run N/256 blocks of 256 threads each
    vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

GPU Memory Hierarchy

- Accessing data from main RAM is slow
  - Data is copied from host to GPU, worked on, then copied back to host

- GPU has different types of memory:
  - Per-thread registers (read/write, very fast)
  - Per-thread local memory (read/write, low-latency)
  - Per-block shared memory (read/write, low-latency)
  - Per-grid global memory (read/write, slower)
  - Per-grid constant memory (read only, low-latency)



OpenACC

- More science, less programming
  - Simply insert hints (pragma directives) in C or Fortran code and core will run on the GPU
  - Similar model to OpenMP
- Simplifies parallel programming on heterogeneous CPU/GPU systems
  - Demo: https://www.youtube.com/embed/_do2Dwa29EM
  - 78s on single-threaded CPU to 6.8s when also using the on-board GPU

OpenACC example

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    #pragma acc kernels
    {
    #pragma acc loop
    for ( int j = 1; j < n-1; j++ ) {
        for ( int i = 1; i < m-1; i++ ) {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] +
                             A[j-1][i] + A[j+1][i] );
                error = fmax ( error,
                               fabs(Anew[j][i] - A[j][i]) );
            }
        }
        #pragma acc loop
        for ( int j = 1; j < n-1; j++) {
            for (int = i; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }
    if (iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
    iter++;
}
```

Many-core CPUs

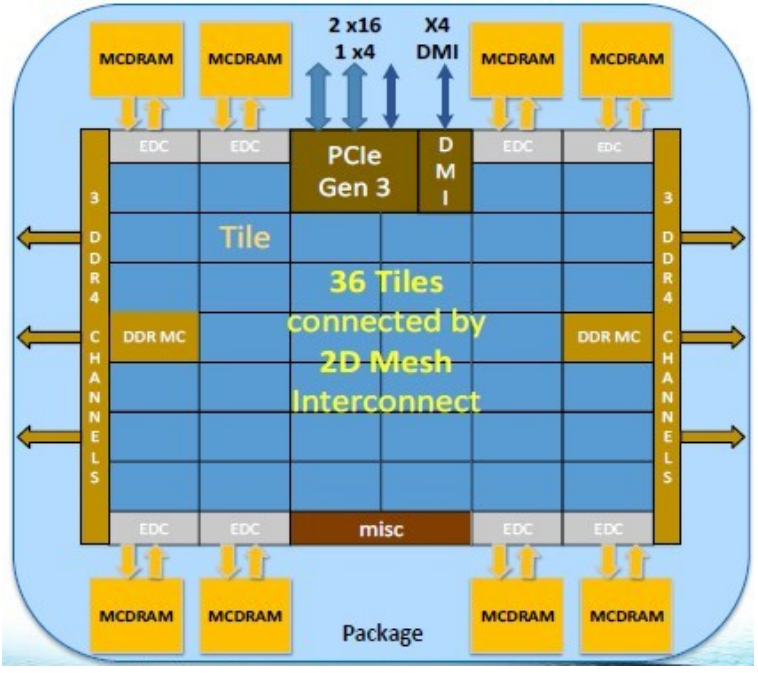
1997: THE FIRST INTEL® TERAFLOP COMPUTER
consisted of: 9,298 INTEL PROCESSORS
and occupied: 72 SERVER CABINETS

THE INTEL® XEON® PHI™ COPROCESSOR
will provide: 1 TERAFLOP OF PERFORMANCE
and occupy: 1 PCIe SLOT

intel inside™ Xeon Phi™

Knights Landing:

- Chip: 36 Tiles interconnected by 2D Mesh
- Tile: 2 Cores + 2 VPU/core + 1Mb L2 cache
- Memory:
  - MCDRAM: 16 Gb on-package, High BW
  - DDR4: 6 channels, up to 384Gb
- Fabric: Omni-path on-package



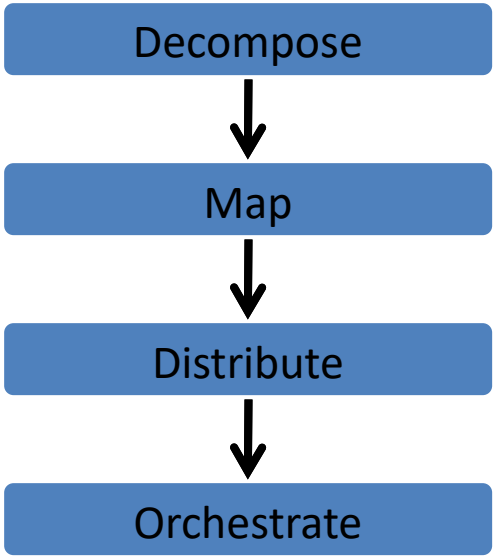
Many-core CPUs

- MIMD (Many instructions many data)
- Supercomputer on a die
  - Slower cores, but lots of them
  - Reduced instruction set
- Similar to multi-core, but
  - Communicate instructions via mesh channels
  - More complicated memory hierarchy
- Keep all cores busy to get good performance
  - Load balance

MESSAGE PASSING DISTRIBUTED COMPUTING

Elements of a Parallel Algorithm

- Pieces of work that can be done concurrently
  - tasks
- Mapping of the tasks onto multiple processors
  - processes *vs* processors
- Distribution of input/output & intermediate data across the different processors
- Management of access to shared data
  - either input or intermediate
- Synchronization of the processors at various points of the parallel execution



Holy Grail:

Maximize concurrency and reduce overheads due to parallelization!
Maximize potential speedup!

Finding Concurrent Pieces of Work

- Decomposition:
  - The process of dividing the computation into smaller pieces of work i.e., *tasks*
- Tasks are programmer defined and are considered to be indivisible.
- Tasks can be of different size.
  - *granularity of a task*

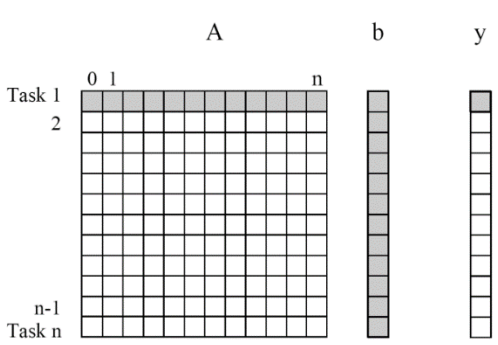


Figure 3.1 Decomposition of dense matrix-vector multiplication into n tasks, where n is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

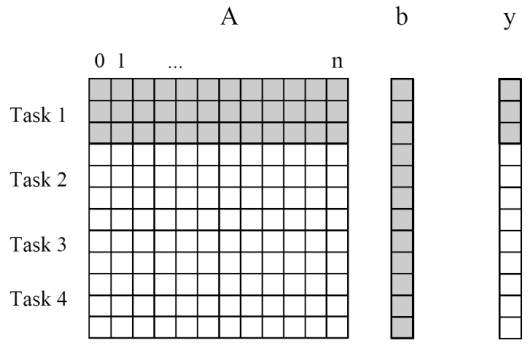


Figure 3.4 Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

MPI: Message Passing Interface

- Standard API for portable parallel programming via message passing
  - Processes share data and work by sending/receiving messages to/from other processes
  - Several efficient implementations in C, C++, Fortran
- Communication model:
  - Shared memory/OpenMP: read/write the same memory location
  - GPGPU: only with other threads in same block, read/write the same block or global shared memory location
  - Distributed/MPI: send/receive messages

Graph Partitioning

- Task mapping can be achieved by directly partitioning the task interaction graph.
  - EG: Finite element mesh-based computations

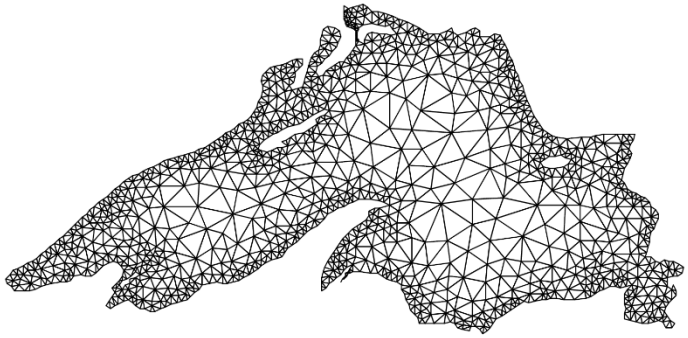


Figure 3.34 A mesh used to model Lake Superior.

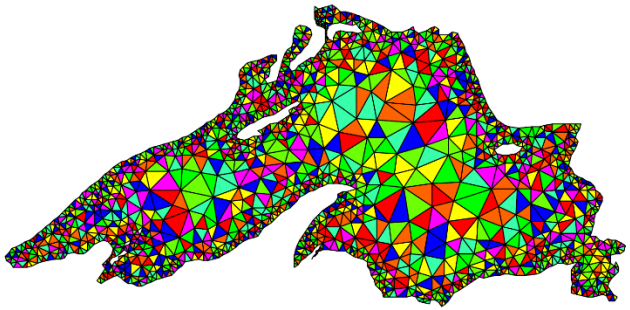


Figure 3.35 A random distribution of the mesh elements to eight processes.

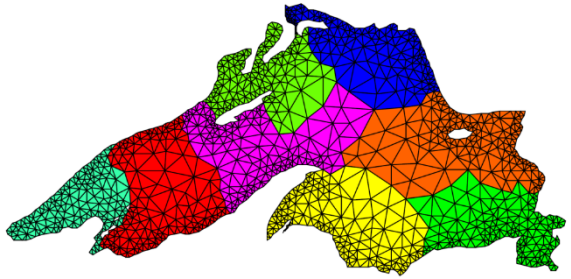


Figure 3.36 A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm.

SHARE-NOTHING DISTRIBUTED COMPUTING

What is Apache Hadoop?

- Open source software framework designed for storage and processing of large scale data on clusters of commodity hardware

- Created by Doug Cutting and Mike Carafella in 2005.

- Cutting named the program after his son's toy elephant.

Slides from Daniel Leblanc, Portland State: web.cecs.pdx.edu/~dleblanc/hadoop/hadoop.pptm

Uses for Hadoop

- Data-intensive text processing
- Assembly of large genomes
- Graph mining
- Machine learning and data mining
- Large scale social network analysis

The Hadoop Ecosystem

Hadoop Common	• Contains Libraries and other modules
HDFS	• Hadoop Distributed File System
Hadoop YARN	• Yet Another Resource Negotiator
Hadoop MapReduce	• A programming model for large scale data processing

Slides from Daniel Leblanc, Portland State: web.cecs.pdx.edu/~dleblanc/hadoop/hadoop.pptm

Motivations for Hadoop

- What were the limitations of earlier large-scale computing?
  - Programming on a distributed system is complex
    - Synchronizing data exchanges
    - Managing a finite bandwidth
    - Controlling computation timing is complicated
  - Not designed around expectation of failure
  - Outdated data storage model:
    - Typically divided into Data Nodes and Compute Nodes
    - At compute time, data is copied to the Compute Nodes

How much data?

- Facebook
  - 500 TB per day
- Yahoo
  - Over 170 PB
- eBay
  - Over 6 PB

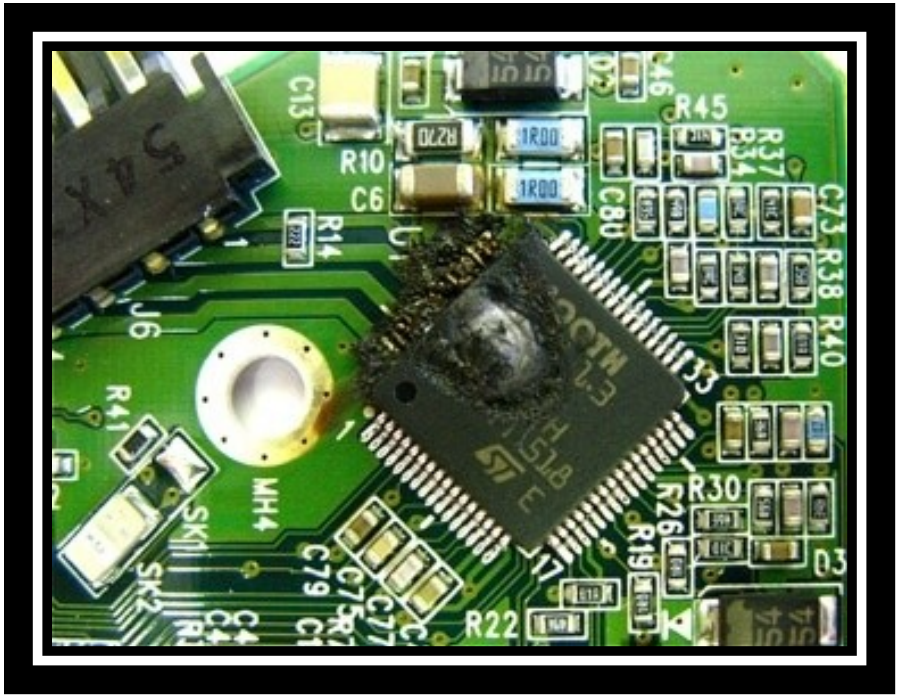
- Getting the data to the processors becomes the bottleneck

Requirements for Hadoop

- Must support partial failure
- Must be scalable

Partial Failures

- Failure of a single component must not cause the failure of the entire system only a degradation of the application performance
  - Failure should not result in the loss of any data



Component Recovery

- If a component fails, it should be able to recover without restarting the entire system
- Component failure or recovery during a job must not affect the final output

Scalability

- Increasing resources should increase load capacity

- Increasing the load on the system should result in a graceful decline in performance for all jobs
  - Not system failure

Hadoop

- Based on work done by Google in the early 2000s
  - "The Google File System" in 2003
  - "MapReduce: Simplified Data Processing on Large Clusters" in 2004
- The core idea was to distribute the data as it is initially stored
  - Each node can then perform computation on the data it stores without moving the data for the initial processing

Core Hadoop Concepts

- Applications are written in a high-level programming language
  - No network programming or temporal dependency
- Nodes should communicate as little as possible
  - A “shared nothing” architecture
- Data is spread among the machines in advance
  - Perform computation where the data is already stored as often as possible

High-Level Overview

- When data is loaded onto the system it is divided into blocks
  - Typically 64MB or 128MB
- Tasks are divided into two phases
  - Map tasks which are done on small portions of data where the data is stored
  - Reduce tasks which combine data to produce the final output
- A master program allocates work to individual nodes

Fault Tolerance

- Failures are detected by the master program which reassigns the work to a different node
- Restarting a task does not affect the nodes working on other portions of the data
- If a failed node restarts, it is added back to the system and assigned new tasks
- The master can redundantly execute the same task to avoid slow running nodes

Other Tools

- Hive
  - Hadoop processing with SQL
- Pig
  - Hadoop processing with scripting
- Cascading
  - Pipe and Filter processing model
- HBase
  - Database model built on top of Hadoop
- Flume
  - Designed for large scale data movement
- Spark
  - In-memory MapReduce via resilient distributed datasets (RDD) – restricted distributed shared memory

Summary

- How to scale data mining
  - Assume sample is similar to population
  - Use approximate algorithms
  - Use parallel computing
- Different types of parallel computing depending on
  - Architecture
  - Task decomposition
  - Communication
- CMPE 213: Parallel Computing
  - Coming next Spring