Crash Course in Map Reduce and GPU Programming

Rice University

Anshumali Shrivastava

anshumali At rice.edu

15th Janauary 2016

The Headaches of Programming over Clusters

- Scheduling Processes.
- Fault Tolerance.
- Load Balancing.
- Synchronization.
- Minimize Communication.
- Data Locality.
- etc.

Parallel Data Processing Simplified: MapReduce

MapReduce is a programming model, by Google, which essentially abstracts away all these headaches of parallelism for a generic aggregation task

- Users specify computation in the form of **Map** and **Reduce** functions (example coming)
- Underlying system automatically parallelizes the computations.
- Performance issues of parallelism are handled by the framework.
- Ideally designed for aggregation task, such as counting (Aggregation forms the core of many algorithms, specially ML and data mining algorithms). So its useful.

Programming Example

Problem: Count the word frequencies over web documents.

Programming Example

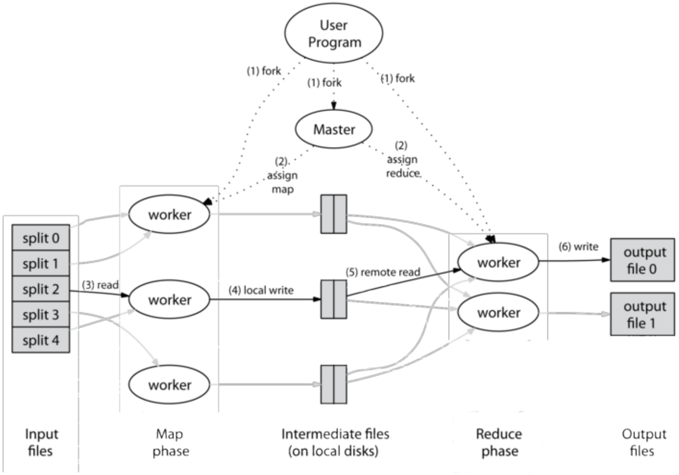
Problem: Count the word frequencies over web documents.

User Implements only Two Functions:

```
function map(String name, String document):
// name: document name
// document: document contents
for each word w in document:
   emit (w, 1)
```

```
function reduce(String word, Iterator partialCounts):
// word: a word
// partialCounts: a list of aggregated partial counts
sum = 0
for each pc in partialCounts:
   sum += pc
emit (word, sum)
```

Workflow[1]



[1]Image from MapReduce: Simplified Data Processing on Large Clusters" by Jeffrey Dean and Sanjay Ghemawat, OSDI'04

How Does it Work?

- Input Reader: Splits data and assign different splits to the map function.
- **Map Function:** Takes series of key/value pairs, processes each, and generates zero or more output key/value pairs.
- Partition Function: The partition function is given the key and the number of reducers and returns the index of the assigned reducer. Usually index = key mod $|Reducers|$
- Comparison Function: The input for each Reduce is taken from the machine where the Map ran and sorted.
- **Reduce Function:** Called once for each unique key in the sorted order. The Reducer takes key and list of associated values can produce zero or more output.
- Output Writer:

Key Performance Considerations

- Between the map and reduce stages, the data is shuffled (parallel-sorted / exchanged between nodes) though by a highly optimized function. This can be longer depending on what map produces.
- Many MapReduce implementations are designed to write all communication to distributed storage for crash recovery. Communication cost can affect the performance
- The amount of data produced by the mappers decides the computation cost that happens between mapping and reducing. Small partition size is preferred (Why ?) but not too small as other overheads come in.
- If data fits in memory, MapReduce is ineffective.

Popular Implementations on MapReduce

- Pattern-Based Searching.
- Distributed Sorting.
- Web Links or PageRank.
- SVD.
- Indexing.
- Machine Translation.
- Machine Learning.

Programming with GPUs

GPUs

- Thousands of cores! Compare this with CPUs which have only few.
- Ideal, when we want to apply the same operation many times at different locations. (example)
- Originally used for graphics operations like
  for all pixels (i,j): replace previous color with new color according to rules

Example Program

Problem: Add two giant arrays C[] = A[] + B[]

CPU

float *C = malloc(N * sizeof(float));
for (int i = 0; i ¡ N; i++)
C[i] = A[i] + B[i];
Muli-core can give us 2-8x speed with tricks.

On GPUs

(allocate memory for A, B, C on GPU)
Create the kernel each thread will perform one (or a few) additions
Specify the following kernel operation:
For (all i's assigned to this thread)
C[i] = A[i] + B[i]
Start 100000 (!) threads

Steps

- Setup inputs on the host (usual malloc)
- Allocate memory for inputs on the GPU
- Allocate memory for outputs on the host (usual malloc)
- Allocate memory for Outputs on the GPU
- Copy inputs from host to GPU
- Start GPU kernel
- Copy output from GPU to host

How the Code Looks Like

```
int * c CUDAadd(float * a, float * b, float * c)
{ double * da, * db, * dc
sizet bytes = n* sizeof(double);
cudaMalloc(& da, bytes);
cudaMalloc(& db, bytes);
cudaMalloc(& dc, bytes);
cudaMemcpy( da, a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( db, b, bytes, cudaMemcpyHostToDevice);
int blockSize, gridSize;
blockSize = 1024;
gridSize = (int)ceil((float)n/blockSize);
vecAdd<<<gridSize, blockSize>>>(da, db, dc, n);
cudaMemcpy( c, dc, bytes, cudaMemcpyDeviceToHost );
cudaFree(da);
cudaFree(db);
cudaFree(dc);
}
```

CUDA Kernel

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
// Get our global thread ID
int id = blockIdx.x*blockDim.x+threadIdx.x;
if (id < n)
c[id] = a[id] + b[id];
}
```

Some Performance Considerations

- Ideal for Matrix or Tensor Multiplication.
- Memory transfer between host and device is slow.
- Actual processing is slower than CPU.
- CUDA will not work unless you exploiting lots and lots of parallelism.

Next Week : Streaming and Sketching