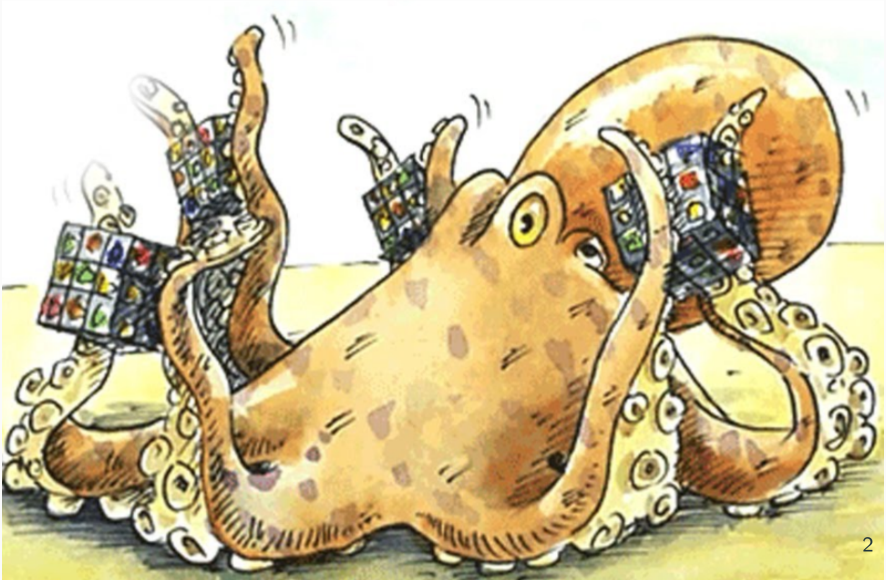
Jacobi method Implementation in Spark

Azam Asl

September 28, 2017

Computer Science Department

Parallel computing!



Project overview

- We implemented Jacobi method to solve a Laplace equation in Spark + YARN in three different implementation.

Project overview

- We implemented Jacobi method to solve a Laplace equation in Spark + YARN in three different implementation.
- First implementation is a naive implementation using dense matrix. It didn't scale for large discretizations ($N \geq 10^5$).

Project overview

- We implemented Jacobi method to solve a Laplace equation in Spark + YARN in three different implementation.
- First implementation is a naive implementation using dense matrix. It didn't scale for large discretizations ($N \geq 10^5$).
- The second implementation which uses sparse matrix scales at least 10 times N in compare to our first implementation.

Project overview

- We implemented Jacobi method to solve a Laplace equation in Spark + YARN in three different implementation.
- First implementation is a naive implementation using dense matrix. It didn't scale for large discretizations ($N \geq 10^5$).
- The second implementation which uses sparse matrix scales at least 10 times N in compare to our first implementation.
- The third implementation is an improvement on second version and it runs faster on my laptop.
- Testing on YARN cluster for $N > 10^6$ was useless. Supposedly because of unresolved issue that Spark application hangs when dynamic allocation is enabled in Hadoop enviroment.

Outline

- Problem statement.
- Implementation details.
- Spark UI demo.

Problem statement: 1D Laplace equation (aka. boundary value problem)

Given a function $f : [0,1] \to \mathbb{R}$ we would like to find function u such that its second derivative within the interval $(0,1)$ is equal to $-f$ and its boundary values are 0. More precisely:

$$u^{''} = -f \text{ in } (0,1) \text{ and } \; u(0) = 0, \; u(1) = 0$$

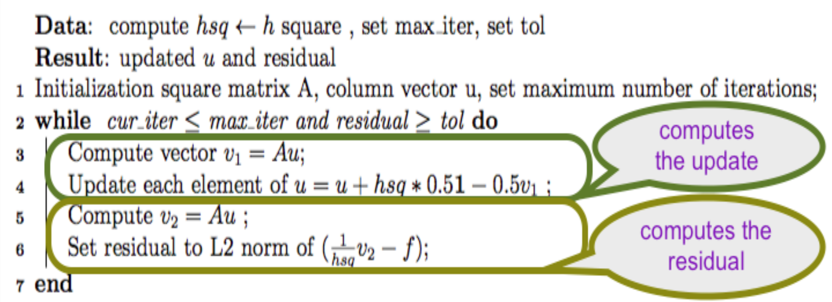
Using finite-difference approximations for the second derivative to approximate the solution, Using Taylor expansions of $u(x_i - h)$ and $u(x_i + h)$ where : $\{x_i = ih : i = 0, 1, \dots, N, N+1\} \subset [0,1]$, with $h = \frac{1}{N+1}$:

$$\frac{1}{h^2} \overset{\text{A}}{\begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{bmatrix}} \overset{\text{u}}{\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix}} = \overset{\text{f}}{\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix}}$$

Solving Laplace equation with Jacobi

In every iteration computes u^{k+1} (i.e. update) from u^k per below (element-wise):

$$u_i^{k+1} = \frac{1}{a_{ii}}\left(f_i - \sum_{j \neq i} a_{ij} u_j^k\right)$$



Data: compute $hsq \leftarrow h$ square , set max_iter, set tol

Result: updated u and residual

Initialization square matrix A, column vector u, set maximum number of iterations;

while *cur_iter* $\leq$ *max_iter and residual* $\geq$ *tol* **do**

  Compute vector $v_1 = Au$;

  Update each element of $u = u + hsq * 0.51 - 0.5v_1$;

  Compute $v_2 = Au$;

  Set residual to L2 norm of $(\frac{1}{hsq}v_2 - f)$;

end

DUMBO: NYU's YARN Cluster

- 48 total nodes, running Cloudera CDH 5.9.0 (Hadoop 2.6.0 with YARN)
- Two of the nodes are the Master nodes (babar and hathi).
- Two other nodes are the login nodes (dumbo0 and dumbo1)
- That leaves us with 44 compute nodes.
- Each node has 128 GB RAM.
- Each node has two CPUs. Each CPU is a 8-core Intel 'Haswell' (c 2014).

Spark+YARN environment settings

- **'–executor-cores'** : Tells Spark to assign each executor 5 corse. Each executor is a JVM instance and will run 5 concurrent tasks.

Spark+YARN environment settings

- **'–executor-cores'** : Tells Spark to assign each executor 5 corse. Each executor is a JVM instance and will run 5 concurrent tasks.
- **'–num-executors'** : This is the number of nodes in abstract. $15/5 = 3$ so $3 \times 43 = 129$ executors.

Spark+YARN environment settings

- **'–executor-cores'** : Tells Spark to assign each executor 5 corse. Each executor is a JVM instance and will run 5 concurrent tasks.
- **'–num-executors'** : This is the number of nodes in abstract. $15/5 = 3$ so $3 \times 43 = 129$ executors.
- **'–executor-memory'**: We have 127 GB RAM and 3 executors per node $\rightarrow$ each executor gets $\sim$ 42 GB of RAM. Accounting for $\sim$ 10% overhead memory $\rightarrow$ each executor get 38 GB of RAM .

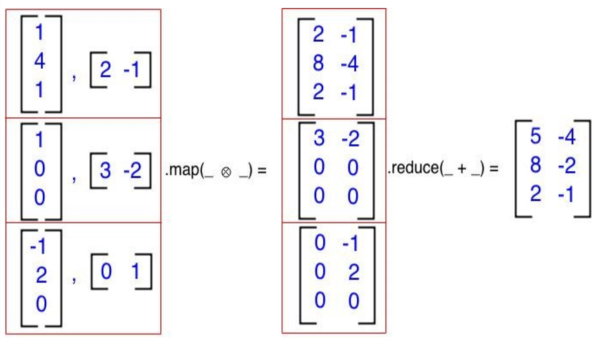
First implementation: Using Spark's Distributed

- Use Saprk's (MLlib) distributed Matrix, specifically BlockMatrix and CoordinateMatrix to implement A and u as dense matrices.

Second implementation : Using scalable matrix multiplication

$$\underbrace{\begin{bmatrix} 1 & 1 & -1 \\ 4 & 0 & 2 \\ 1 & 0 & 0 \end{bmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{bmatrix} 2 & -1 \\ 3 & -2 \\ 0 & 1 \end{bmatrix}}_{\mathbf{B}} = \underbrace{\begin{bmatrix} 5 & -4 \\ 8 & -2 \\ 2 & -1 \end{bmatrix}}_{\mathbf{C}}$$

Co-group columns of A with rows of B and perform a outer product on each group. Reduce by row to compute C:



Second implementation

```
Computing A*u_k
val Auk_rdd = A_rdd.join(u_rdd)
        .flatMap{case(k, v) => v._1.map(mv => (mv._1, mv._2 * v._2))}
        .reduceByKey(_ + _)
```

Third implementation: Don't move A, broadcast u

```
var u_recie = sc.broadcast(u_rdd.collect().toMap).value
```

Now, each core has a complete copy of u. Compute Au with one less shuffling:

```
Auk_rdd = A_rdd
        mapPartitions({ iter =>
         for {(ak, av) <- iter
           } yield (ak, (av, u_recie.get(ak).get))
         },preservesPartitioning = true)
        .flatMap{case(k, v) => v._1.map(Av => (Av._1, Av._2 * v._2))}

        .reduceByKey(_ + _).cache()//reduce on rows
```

Third implementation: more improvements

- Avoid redundant computation of Au^{k+1} : Better solution: compute Au^{k+1} in kth iteration and cache it for the next iteration.

Third implementation: more improvements

- Avoid redundant computation of Au^{k+1} : Better solution: compute Au^{k+1} in kth iteration and cache it for the next iteration.
- Use accumulator to compute the residual: Accumulators are added to through an associative and commutative operation and therefore are efficient.

Third implementation: more improvements

- Avoid redundant computation of Au^{k+1} : Better solution: compute Au^{k+1} in kth iteration and cache it for the next iteration.
- Use accumulator to compute the residual: Accumulators are added to through an associative and commutative operation and therefore are efficient.
- Increase 'numTasks' for reduceByKey operations (default is 8).

Third implementation: every iteration:

```
//        Computing u_{k+1} = 0.5*hsq-0.5A*u_k+u_k
           u_rdd = Auk_rdd.mapValues(hsqHalf - 0.5 * _)
                 .mapPartitions({ iter =>
                   for {(ak, av) <- iter
                     } yield (ak, (av, u_recie.get(ak).ge
                   },preservesPartitioning = true)
                 .mapValues{ s =>s._1 + s._2}.cache()
           u_recie = sc.broadcast(u_rdd.collect().toMap).value
//         Computing A*u_{k+1}
           Auk_rdd = A_rdd
                  .mapPartitions({ iter =>
                    for {(ak, av) <- iter
                      } yield (ak, (av, u_recie.get(ak).get))
                    },preservesPartitioning = true)
                  .flatMap{case(k, v) => v._1.map(Av => (Av._1, Av._2 * v._2))}
                  .reduceByKey(_ + _,numTasks).cache()//reduce on rows

//         Computing residual
           val accum = sc.doubleAccumulator("Accumulating re
           Auk_rdd.mapValues(invhsq*_ - 1)
                  .map{ case(k, v) => v * v}
                  .foreach(x => accum.add(x))
          ress(k) = Math.sqrt(accum.value)
          k+=1
```

update u: without shuffling.

broadcast u

Compute new Au and cache it

use accumulator for residual

Third implementation: Spark UI view

- Every SparkContext launches a web UI, by default on port 4040.
- Spark can only run one concurrent task for every partition of an RDD, up to the number of cores (either physical or spark cores).
- Running previous code for N = 50000, 1 iteration and 5 partitions on my laptop : master("local[*]")

Completed Jobs (3)

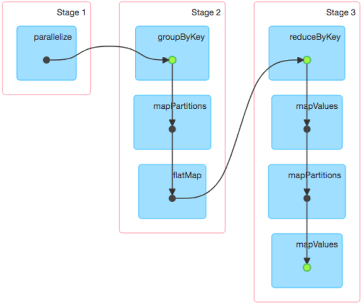
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	foreach at Jacobi.scala:70	2017/05/15 19:25:49	0.4 s	2/2 (1 skipped)	10/10 (5 skipped)
1	collect at Jacobi.scala:56	2017/05/15 19:25:47	2 s	3/3	15/15
0	collect at Jacobi.scala:38	2017/05/15 19:25:45	0.5 s	1/1	8/8

Third implementation: Spark UI view

Details for Job 1

Status: SUCCEEDED
Completed Stages: 3

- Event Timeline
- DAG Visualization



Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	collect at Jacobi.scala:56 +details	2017/05/15 19:25:49	0.4 s	5/5			599.7 KB	
2	flatMap at Jacobi.scala:45 +details	2017/05/15 19:25:47	1 s	5/5			1236.3 KB	599.7 KB
1	parallelize at Jacobi.scala:32 +details	2017/05/15 19:25:47	0.4 s	5/5				1236.3 KB