THE EFFECTS OF DATA LOCALITY ON DATA INTENSIVE COMPUTING

SANJAY AGRAVAT
CS 584 - HIGH PERFORMANCE COMPUTING

Abstract. Data Intensive Computing presents many challenges with parallel computing including storage, managing, accessing, and processing vast amounts of data. In this paper, we discuss the effects of data locality on Data Intensive Computing by comparing an MPI implementation with a Map Reduce (Hadoop) implementation. We show that partitioning the data with a context-aware mapping provides significant performance improvement compared to a Map Reduce approach which have no data locality considerations in their reduce tasks.

Date: December 6, 2011.

1. Introduction

Biomedical Informatics involves the processing of enormous amounts of data ranging from clinical health records to Next Gen Sequencing data to digital pathology slides. These applications naturally fall under the Data Intensive Computing paradigm. In this paper, we will demonstrate how data locality impacts the performance of Data Intensive Computing using an application that comes from Digital Pathology Imaging.

Digitized pathology images contain millions of nuclei that exhibit various different morphological characteristics. Morphology characteristics can reveal genetic alterations and are predictive of patient prognosis and treatment response.

The nuclear analysis workflow, shown in fig 1, includes three steps. The nuclei segmentation is used to automatically segment nuclei on the slide. The next step is feature extraction which is where you calculate things such as shape and area, and lastly the nuclei classification which is used to assign labels to the different subtypes of gliomas. Our application is only concerned with the nuclei segmentation step.

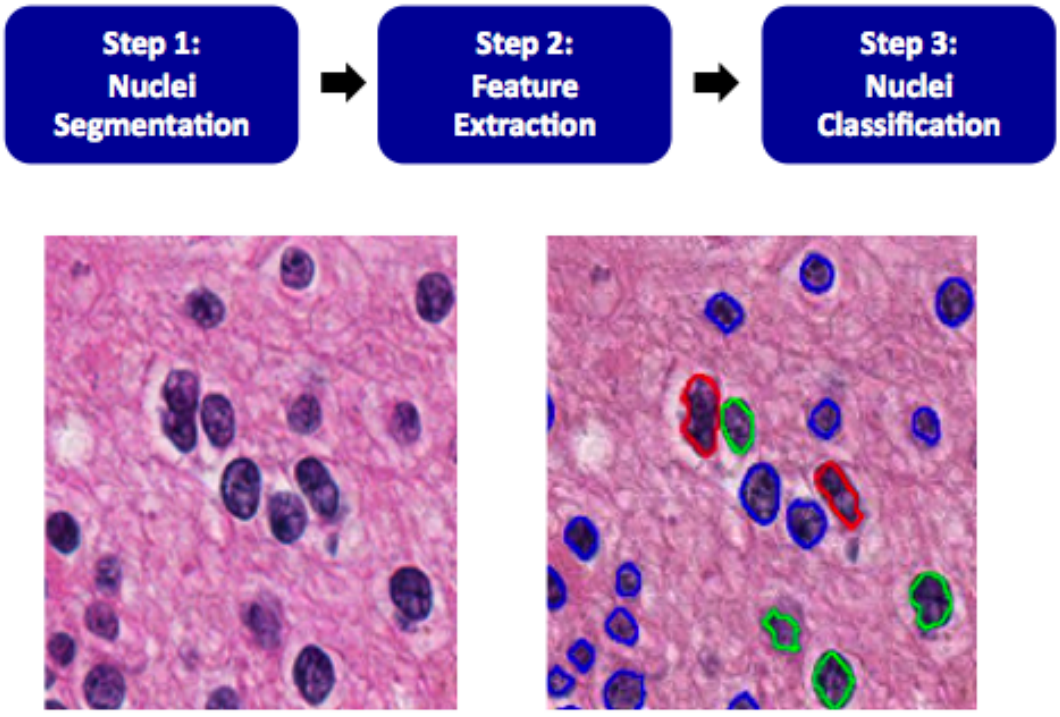

Figure 1. Nuclear Analysis Workflow

In particular, we want to support algorithm validation. That is, we have 2 or more algorithms for doing nuclei segmentation and we would like to perform a comparison on the differences between the two algorithms' segmentation results.

Fig, 2 shows an example of a region that is segmented by two different algorithms, one is red and the other is green. Notice how some of the segmentations overlap and some do not exist for the red ones or vice versa. Our objective is to perform a spatial join operation on these nuclei (or more generally, polygons) to determine how much overlap or similarity there is between the different algorithms.

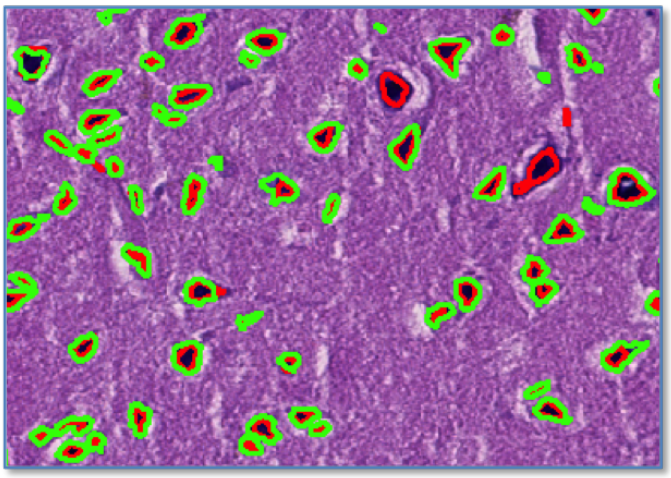

FIGURE 2. Nuclear Segmentation of two Algorithms

Rather than doing an n^2 comparison of all polygons, we create an R Tree index to reduce the search space. The index is built is as follows: (a) First, a Minimum Bounding Box for each polygon is created. (b) Next, the Minimum Bounding Boxes that overlap are grouped together, as well as other ones in close proximity such that a balance condition is met to minimize the area and number of bounding boxes. (c) The clustered bounding boxes are also grouped together. (d) Finally, the bounding boxes are grouped together using a parameter, M, which is the maximum number of bounding boxes you can have at any node in the RTree. This type of indexing allows for faster querying of spatial data such that it takes log N time to find all the polygons that share a bounding box with each other. The spatial join is done on those polygons or nuclei using the Computational Geometry Algorithm Library (CGAL) written in C++.

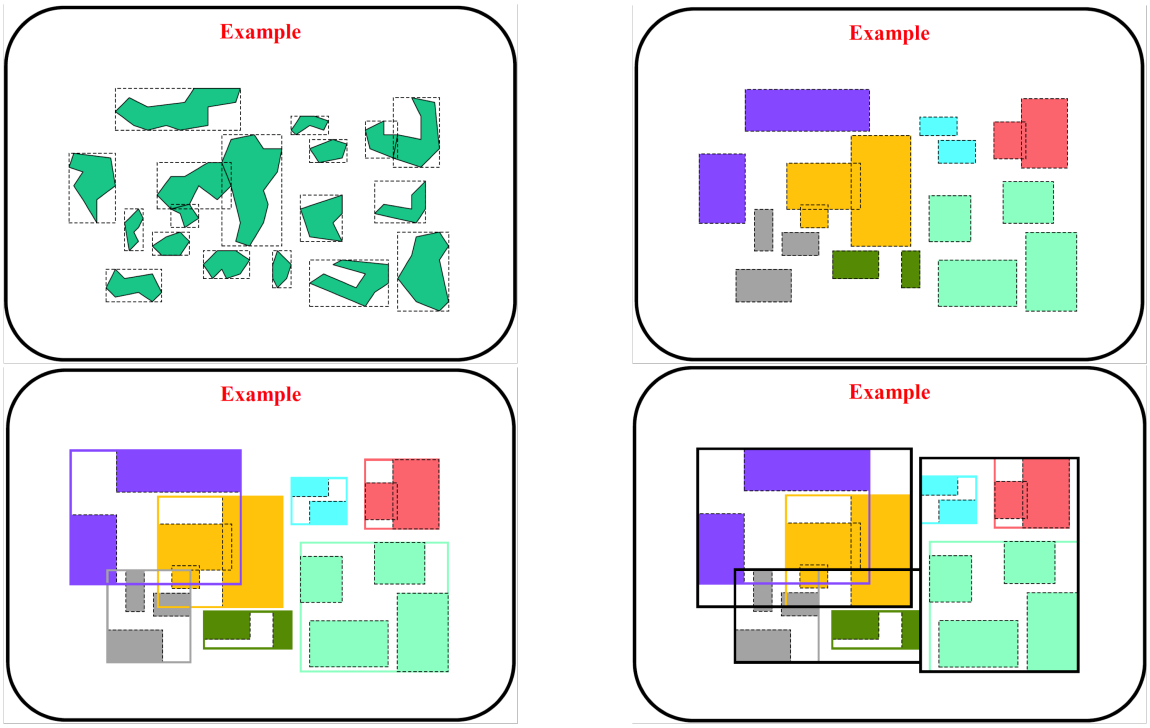

FIGURE 3. R*Tree Indexing

2. RESULTS

The input data set consists of 11.7GB of approximately 4,000 text files that contain the tile ids and the polygon coordinates. Each file is named in a conventional format to indicate the tile id and the algorithm id used to create the tile image. In terms of how this application fits into Map Reduce, the Map phase groups those tile ids into keys and checks if there is a corresponding tile id for the other algorithm. If the tile exists then it adds it as a key while discarding records that don't have a corresponding key.

The reduce phase dominates the work load; taking about 90% of total processing time. The reducers job is to build the spatial index for the tile, write it to disk, invoke the spatial join operation on a pair of indexes, and finally write the result of the spatial join to disk. Since both the Spatial Index and CGAL library are written in C++, it has to fork a process to execute them.

The MPI design takes advantage of the fact we know we will only do a spatial join operation on the pair of tiles for each algorithm. Thus, we partition all the slides in a preprocessing step from the distributed file system to local storage on each of the local nodes. I explored two approaches for partitioning: one was binpacking and the other one

was to divide the dataset size by the number of nodes. The latter approach surprisingly performed better.

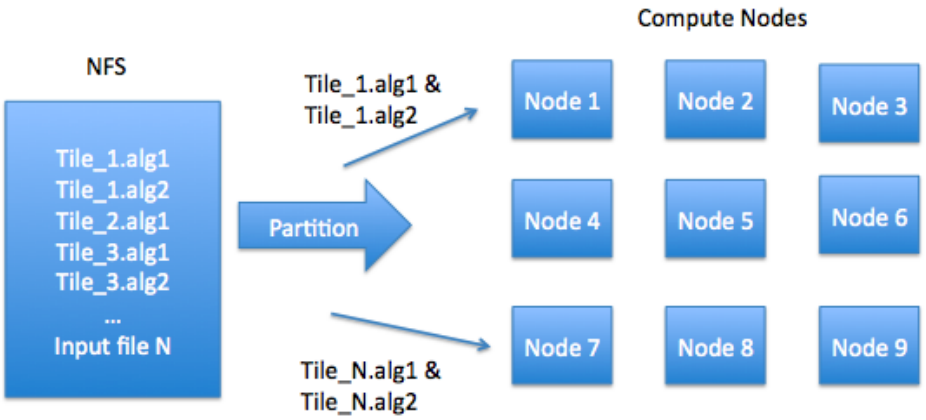

FIGURE 4. MPI Design

Each node has 1 master and *n-1* workers. The data set is written to the same directory for each of the nodes so we don't need to have a mapping file for the metadata. The master on each node recursively traverses the directories and sends a tile to each worker, the worker processes the tile by creating the spatial index and then it performs the spatial join operation on the index pair and writes it to disk.

2.1. **Experimental Setup.** We have two configurations used for the experiments as shown in Table 1. Configuration 1 is a head-to-head comparison between the MPI Implementation and Hadoop. Both are using a Large Number of Small Files (LNFS) which is a worst case scenario for both systems. This is due to the I/O overhead and the associated overhead with creating Tasks for each input file. It would be more efficient to concatenate the files within a small amount of the block size for more efficiency with Hadoop. A similar design could be considered for the MPI version as well.

Experimental Setup		
	MPI	Hadoop
Configuration 1	9 Nodes 176 Processes 9 Masters 167 Workers LNSF	9 Nodes 100 Reducers Block Storage 128mb JVM Reuse Enabled LNSF
Configuration 2	9 Nodes 176 Processes 9 Masters 167 Workers LNSF	10 Nodes 100 Reducers Block Storage 128mb JVM Reuse Enabled LNSF, HAR, & Merged Files

We see in Fig 5 that the spatial join operation comparing Hadoop with 100 reducers to MPI shows that MPI takes 41% less time which is significantly less than hadoop. The only notable difference between the two systems in this setup is the number of reducers for Hadoop. There were only 100 reducers configured but there were 176 processors available so it would have been useful to do a comparison with approximately 200 reducers.

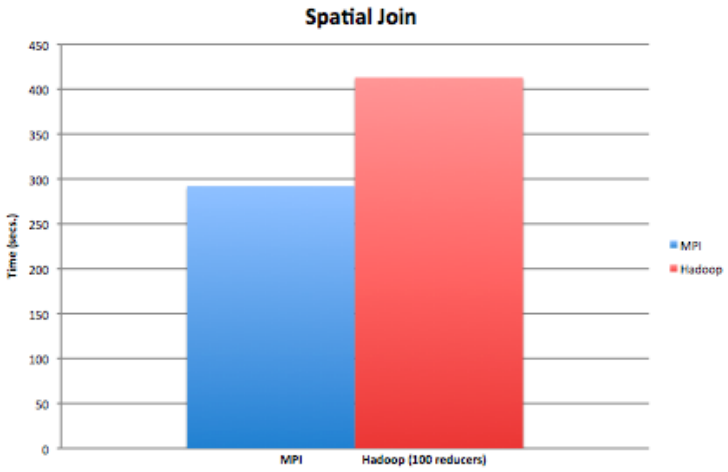

FIGURE 5. Spatial Join - MPI vs. Hadoop

Fig 6 shows the performance results of both Hadoop and MPI when you scale the data set set 3x, and 5x. It shows linear speedup for both systems. The Hadoop version performs better but that is because it was using 10 nodes and also using merged files instead of a large number of small files unlike MPI.

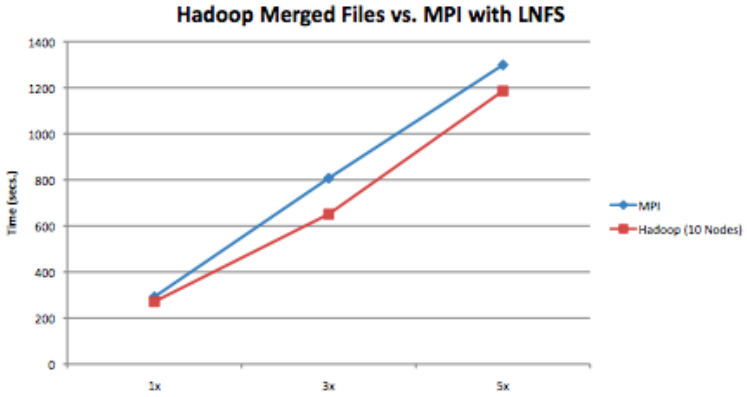

FIGURE 6. Spatial Join - MPI vs. Hadoop

The chart in Fig 7 shows the performance results of the join time vs. the number of reducers which takes into consideration the scaling of the dataset size. It shows that the number of reducers has an effect on the performance as you increase the dataset size up to 200 reducers. There was no data beyond 200 reducers but I imagine it would not increase the performance since only have 192 cores in the cluster

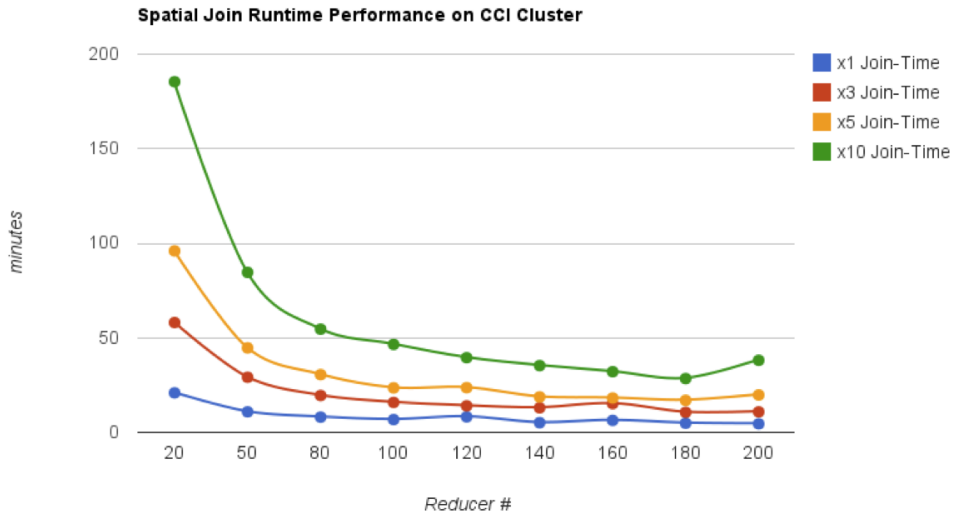

FIGURE 7. Spatial Join Runtime vs. Number of Reducers by data size

Fig 8 demonstrates the effect of using a LNFS, vs. merged files vs. HAR on Hadoop. It shows that HAR does not perform as well as the other two but this could be due to the increase in the overhead in managing HAR files for a small number of files (approx 4000 tiles).

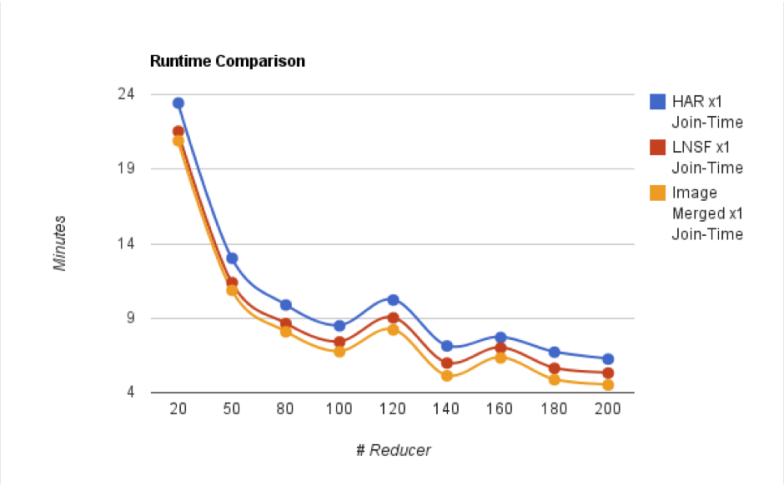

FIGURE 8. Hadoop LNFS vs HAR vs Merged Files

3. CONCLUSION

Two methods were tested and the results demonstrate that data locality plays a large role in performance for Data Intensive Computing applications. Further analysis should be performed to examine the metrics from Hadoop to investigate the amount of data shuffling between nodes required for the Reducers. Future improvements on the MPI implementation include support for merged files and integration with GPU accelerators for Spatial Indexing and Spatial Join operations.