Parallelizing Affinity Propagation using GPUs for Spatial Cluster Analysis over Big Geospatial Data

Xuan Shi

Department of Geosciences, 216 Ozark Hall
University of Arkansas, Fayetteville, AR 72701, USA
Telephone: 479-575-3355
Email: xuanshi@uark.edu

Abstract

Geocomputation has been the foundation of Geoinformatics for knowledge discovery through spatial and temporal data mining and analytics. Traditionally, Geoinformatics software products were developed based on serial computer programs for desktop application. Constrained by the hardware infrastructure and software solutions, geocomputation may not be accomplishable to process and analyze data with increasing scale of data volume and computation complexity. Consequently geoinformatics research will be constrained by the inability of the computational modules in the available software. Emerging computer architectures and systems that combine multicore Central Processing Units (CPUs) and accelerator technologies, like many-core Graphics Processing Units (GPUs) and Intel Many-Integrated Core (MIC) co-processors, could provide the substantial computing power to make breakthrough in geocomputation and geoinformatics research. New multicore and manycore architectures combined with application accelerators hold the promise to achieve scalable geocomputation by exploiting task and data levels of parallelism that are not supported by the conventional systems. Such a distributed and parallel computing environment is particularly suitable for large-scale geocomputation over big data. This paper introduces the recent progress in parallelizing the Affinity Propagation (AP) algorithm on the GPU for spatial cluster analysis, the potential of the proposed solution to process big geospatial data, and the broader impact to the scientific community.

1. Affinity Propagation

Among varieties of classification and clustering approaches for spatial data mining and knowledge discovery (Guo and Mennis 2009), this research targets Affinity Propagation (AP) (Frey and Dueck 2007) for several reasons. The AP algorithm was introduced by *Science* in 2007 and this work (Frey and Dueck 2007) has been cited for 2,300+ by the other researchers. As a relatively new clustering algorithm, AP is not widely applied in geoinformatics yet. Unlike other classification or clustering algorithms, such as ISODATA, k-means, and Maximum Likelihood Classifier, AP does not specify a pre-defined arbitrary number of clusters in advance but will derive the number of clusters as the result. Furthermore, AP can be applied in cluster analysis on raster or image data, vector geometric data, and text data. For this reason, AP has significant potential in geoinformatics in the identification of spatial clusters and other research and applications, such as data resampling, spatial filter, and pattern analysis.

To implement the AP algorithm (Frey and Dueck 2007), a similarity matrix S contains n x (n-1) records of the negative values of the distance between each point to all other points. The other input data contains the preference value of the n input points. The similarity matrix S describes how each data point is presented to be the exemplar, while data points with higher preference values could be selected as cluster centers or exemplars. In this case, the preference value determines the number of identified clusters. In AP, all data points are considered equally as potential exemplars or the cluster centers. For this reason, the preference values are initialized to a common value, which is usually the median in the similarity matrix. In general, AP is an optimization process to maximize the similarity or to minimize the total sum of intra-cluster similarities.

The number of clusters eventually emerges by iteratively passing messages between data points to update two matrices (Frey and Dueck 2007). The "responsibility" matrix R has values r(i, k) that quantify how well-suited point k is to serve as the exemplar for point i relative to other candidate exemplars for point i. The "availability" matrix A contains values a(i, k) represents how "appropriate" it would be for point i to pick point k as its exemplar, taking into account other points' preference for point k as an exemplar. Both matrices A and R are initialized to all zeroes. The AP algorithm then performs the updates iteratively over the two matrices. First, “Responsibilities” r(i,k) are sent from data points to candidate exemplars and indicate how strongly each data point favors the candidate exemplar over other candidate exemplars. “Availabilities” a(i,k) are then sent from candidate exemplars to data points and indicate to what degree each candidate exemplar is available as a cluster center for the data point. In this case, the responsibilities and availabilities are messages that provide evidence for whether or not each data point should be an exemplar and if not to what exemplar that data point should be assigned. For each iteration in the message-passing procedure, the sum of r(k; k) + a(k; k) can be used to identify exemplars. After the messages have converged, there are two ways to identify exemplars. In the first approach, for data point i, if r(i,i)+a(i,i) > 0, then data point i is an exemplar. In the second approach, for data point i, if r(i,i)+a(i,i) > r(i,j)+a(i,j) for all i not equal to j, then data point i is an exemplar. The whole procedure terminates after it reaches a predefined number of iterations, or if the decided clusters have stayed constant for some iterations.

2. Computation Constraints in Affinity Propagation

Although AP has obvious advantages in comparison to many other approaches for clustering analysis (Frey and Dueck 2008), it was acknowledged (Dueck 2009) that “*Affinity propagation’s computational and memory requirements scale linearly with the number of similarities input; for non-sparse problems where all possible similarities are computed, these requirements scale quadratically with the number of data points*.” It took hours or a day to complete the AP calculation over some sample datasets discussed in Dueck’s dissertation (2009).

In geospatial applications, several prior works (Yang et al. 2010, Chehdi, et al. 2014) would only be able to handle a *tiny* datasets as prototypes to test the AP approach, since the images mentioned in the prior publications only had a dimension of several dozens or hundreds of pixels, or the image size allowed by the AP algorithm in MATLAB environment should not exceed 3,000 pixels. In the case of image analytics, an image with a dimension of 100 pixels x 100 pixels has a total of 10,000 pixels. The size of similarity matrix is almost about 10^8 which could hardly be efficiently processed by the serial program of AP. When a single tile of high resolution image could easily contain

10^8~10^9 pixels, the computation could simply go beyond petascale (10^{15}) or exascale (10^{18}). The same scalability and performance constraints exist when large amount of geospatial features in vector datasets are used. If AP can be applied in geoinformatics to resolve real-world problems, the scalability bottleneck has to be overcome. Currently, there is *no* parallel and distributed computing solution yet for AP (AP FAQ).

3. Parallelization of Affinity Propagation

Although the sample data and a C program of affinity propagation (AP) is provided online[1], the parallelization of AP seems not an intuitive and easy process. The toy data has 25 points with x, y coordinated recorded in a text file. The two input text files include a similarity file and a preference file. The similarity file contains the information about the distance for each point to all other points and thus has 25 x 24 = 600 records. For each row, it contains the identification of a given points, the identification of a corresponding point, and the negative value of the distance between the two points. The preference file contains 25 rows with the median value of the distance documented in the similarity file.

	1	2	3	4	5	6	7	8	9
0		2	3	4	5	6	7	8	9
0	1		3	4	5	6	7	8	9
0	1	2		4	5	6	7	8	9
0	1	2	3		5	6	7	8	9
0	1	2	3	4		6	7	8	9
0	1	2	3	4	5		7	8	9
0	1	2	3	4	5	6		8	9
0	1	2	3	4	5	6	7		9
0	1	2	3	4	5	6	7	8	

+

0	1	2	3	4	5	6	7	8	9

Table 1. The index of the input data definition in the C program

Consequently when the input data are read into the program, the index of the input data seems chaotic in comparison to a regular matrix. Within a regular 2D matrix annotated by `n x n` dimension, it is easy to loop through the matrix to complete the calculation. In the C program of AP, however, data is organized in an irregular pattern as 25 x 24 + 25. In a more general format, it is indexed by `n x (n − 1) + n`. To elaborate this problem, a sample of 10 points is used to describe the index of the input data of 10 x 9 + 10 in Table 1. While the input array is indexed from 0 to 99, the value in the array is retrieved from another indexed array as described in Table 1, i.e. [1 2 3 4 5 6 7 8 9 0 2 3 4 5 6 7 8 9 0 1 3 4 5 6 7 8 9 0 1 2 4 5 6 7 8 9 1 2 3 5 6 7 8 9 0 1 2 3 4 6 7 8 9 0 1 2 3 4 5 7 8 9 0 1 2 3 4 5 6 8 9 0 1 2 3 4 5 6 7 9 0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6 7 8 9].

For this reason, a hybrid index approach is applied in the C program. That's to say, the index of the input array is based on the output of the other indexed array. For a given `n`

[1] Source: http://www.psi.toronto.edu/affinitypropagation/apcluster_unsupported.txt

`x n` array, the indexes of the `n x (n - 1)` values, for example, are derived from the other array. For example, two lines of the C program are listed below:

```
for(j=0; j<m-n; j++) if(r[j]>0.0) srp[k[j]]=srp[k[j]]+r[j];
for(j=m-n; j<m; j++)              srp[k[j]]=srp[k[j]]+r[j];
```

In this case, the value of *m* is `n x n`. Although array *k* has a dimension of *m*, the values of *k* are between 0 to *n*. While the dimension of srp is *n*, the index of srp is determined by the value of *k* indexed by *m*. Meanwhile, it is noted that the value of srp is calculated based on different condition when the index of *m* is in different range. Since `(m - n)` is `n x (n - 1)`, when srp is calculated differently by separate data range of `n x (n - 1)` and *n*, which is just `m - (m - n)`, other similar situations in the serial C program of AP have to be handled appropriately and carefully.

By reviewing the implementation details, it can be concluded that breaking the data dependency within the sequential AP program could be the key in re-designing the parallel programs. When the index is applied as `n x (n - 1) + n`, it is difficult to parallelize the serial AP program. For this reason, we need to first re-construct or restore the regular `(n x n)` index framework in order to parallelize the serial C program for AP. As a result, certain modules of the serial C programs have to be decomposed from one module into two modules covering two data ranges of `n x (n - 1)` and *n*.

4. Implementation of Parallelized AP on the Graphics Processing Unit (GPU)

The parallelized AP program has been developed for implementation on the the Graphics Processing Unit (GPU). A sample data of 3,736 points are used to test and validate the CUDA programs using NVIDIA K20 GPU to derive exactly the same result as that generated from the serial AP program. Significant algorithm re-design and reconstruction have to be explored to achieve parallelism. For example, for the above one line of code:

```
for(j=0; j<m-n; j++) if(r[j]>0.0) srp[k[j]]=srp[k[j]]+r[j];
```

the corresponding CUDA programs including both host program and device program are described as the follows:

/*****************CUDA module – host**************************/

```
size_t size2 = (m-n)*sizeof(double);
size_t size3 = n*sizeof(double);

k_h=(unsigned long *)calloc(m-n,sizeof(unsigned long));
r_h=(double *)calloc(m-n,sizeof(double));

for(j=0;j<m-n;j++){
            r_h[j]=r[j];
            k_h[j]=k[j];
}
```

```
cudaMalloc((void **) &r_d, size2);
cudaMalloc((void **) &srp_d, size3);

cudaMemcpy(r_d, r_h, sizeof(double)*(m-n),
cudaMemcpyHostToDevice);
cudaMemcpy(srp_d, srp, sizeof(double)*n, cudaMemcpyHostToDevice);

blockSize = 4;
nBlocks = n/blockSize + (n%blockSize == 0?0:1);

CUDAmodule2 <<< nBlocks, blockSize >>> (n, r_d, srp_d);

cudaMemcpy(srp, srp_d, sizeof(double)*n, cudaMemcpyDeviceToHost);

cudaFree(r_d);
cudaFree(srp_d);


/****************CUDA module – device *************************/

__global__ void CUDAmodule2(unsigned long n, double *r, double
*srp)
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   __syncthreads();

   if (idx<n) {
      for(int j=0;j<n;j++) {
         if(j!=idx){
            if(j==0){
               if(r[(idx-1)+j*(n-1)]>0.0){
                  srp[idx]=srp[idx]+r[(idx-1)+j*(n-1)];
               }
            }
            else{
               if(idx>j-1){
                  if(r[(idx-1)+j*(n-1)]>0.0){
                     srp[idx]=srp[idx]+r[(idx-1)+j*(n-1)];
                  }
               }else{
                  if(r[(idx-1)+j*(n-1)+1]>0.0){
                     srp[idx]=srp[idx]+r[(idx-1)+j*(n-1)+1];
                  }
               }
            }
         }
      }
      __syncthreads();
   }
}
```

When the entire `while` loop is the most time-consuming section in the serial C program, it can be transformed from

```
while(dn==0){
    it++; /* Increase iteration index */

    for(j=0;j<n;j++){ mx1[j]=-MAXDOUBLE; mx2[j]=-MAXDOUBLE; }

      for(j=0;j<m;j++){
      tmp=a[j]+s[j];
      if(tmp>mx1[i[j]]){
           mx2[i[j]]=mx1[i[j]];
           mx1[i[j]]=tmp;
      } else if(tmp>mx2[i[j]]) mx2[i[j]]=tmp;
    }

    for(j=0;j<m;j++){
      tmp=a[j]+s[j];
      if(tmp==mx1[i[j]])
             r[j]=lam*r[j]+(1-lam)*(s[j]-mx2[i[j]]);
      else
             r[j]=lam*r[j]+(1-lam)*(s[j]-mx1[i[j]]);
    }

    for(j=0;j<n;j++) srp[j]=0.0;

    for(j=0;j<m-n;j++) if(r[j]>0.0) srp[k[j]]=srp[k[j]]+r[j];
    for(j=m-n;j<m;j++) srp[k[j]]=srp[k[j]]+r[j];

    for(j=0;j<m-n;j++){
      if(r[j]>0.0) tmp=srp[k[j]]-r[j]; else tmp=srp[k[j]];
      if(tmp<0.0) a[j]=lam*a[j]+(1-lam)*tmp; else a[j]=lam*a[j];
    }
    for(j=m-n;j<m;j++) a[j]=lam*a[j]+(1-lam)*(srp[k[j]]-r[j]);

    decit++; if(decit>=convits) decit=0;
    for(j=0;j<n;j++) decsum[j]=decsum[j]-dec[decit][j];
    for(j=0;j<n;j++)
      if(a[m-n+j]+r[m-n+j]>0.0) dec[decit][j]=1; else
dec[decit][j]=0;
    K=0; for(j=0;j<n;j++) K=K+dec[decit][j];
    for(j=0;j<n;j++) decsum[j]=decsum[j]+dec[decit][j];
    if((it>=convits)||(it>=maxits)){
      conv=1; for(j=0;j<n;j++)
if((decsum[j]!=0)&&(decsum[j]!=convits)) conv=0;
      if(((conv==1)&&(K>0))||(it==maxits)) dn=1;
    }
}
```

to CUDA programs. For comparison purpose, the host programs are listed below:

```
long blockSize = 512;
long nBlocks = m/blockSize + (m%blockSize == 0?0:1);
long nBlocks_n = n/blockSize + (n%blockSize == 0?0:1);

size_t sizeM = m*sizeof(double);
size_t sizeN = n*sizeof(double);
size_t sizeMN = (m-n)*sizeof(double);

cudaMalloc((void **) &mx1_d, sizeN);
cudaMalloc((void **) &mx2_d, sizeN);
cudaMalloc((void **) &a_d, sizeM);
cudaMalloc((void **) &s_d, sizeM);
cudaMalloc((void **) &r_d, sizeM);
cudaMalloc((void **) &i_d, sizeM);
cudaMalloc((void **) &srp_d, sizeN);
cudaMalloc((void **) &k_d, sizeM);
cudaMalloc((void **) &dec_d, sizeof(unsigned long)*n);

cudaMemcpy(mx1_d, mx1, sizeof(double)*n, cudaMemcpyHostToDevice);
cudaMemcpy(mx2_d, mx2, sizeof(double)*n, cudaMemcpyHostToDevice);
cudaMemcpy(k_d, k, sizeof(unsigned long)*m,
cudaMemcpyHostToDevice);
cudaMemcpy(a_d, a, sizeof(double)*m, cudaMemcpyHostToDevice);
cudaMemcpy(s_d, s, sizeof(double)*m, cudaMemcpyHostToDevice);
cudaMemcpy(r_d, r, sizeof(double)*m, cudaMemcpyHostToDevice);
cudaMemcpy(i_d, i, sizeof(unsigned long)*m,
cudaMemcpyHostToDevice);
cudaMemcpy(srp_d, srp, sizeof(double)*n, cudaMemcpyHostToDevice);
cudaMemcpy(dec_d, dec[decit], sizeof(double)*n,
cudaMemcpyHostToDevice);

while(dn==0){
     it++; /* Increase iteration index */
     for(j=0;j<n;j++){ mx1[j]=-MAXDOUBLE; mx2[j]=-MAXDOUBLE;
srp[j]=0.0;}
     CUDAmodule00 <<< nBlocks, blockSize >>> (n, mx1_d, mx2_d,
srp_d);
     CUDAmodule0 <<< nBlocks, blockSize >>> (n, a_d, s_d, mx1_d,
mx2_d);
     CUDAmodule01 <<< nBlocks, blockSize >>> (m, n, a_d, s_d,
mx1_d, mx2_d);
     CUDAmodule1 <<< nBlocks, blockSize >>> (r_d, m, i_d, mx1_d,
mx2_d, a_d, s_d, lam);
     CUDAmodule2 <<< nBlocks, blockSize >>> (n, r_d, srp_d);
     CUDAmodule3 <<< nBlocks, blockSize >>> (m, n, r_d, srp_d);
     CUDAmodule4 <<< nBlocks, blockSize >>> (m, n, r_d, k_d,
srp_d, a_d, lam);
     CUDAmodule5 <<< nBlocks, blockSize >>> (m, n, r_d, k_d,
srp_d, a_d, lam);
     decit++; if(decit>=convits) decit=0;
     for(j=0;j<n;j++) decsum[j]=decsum[j]-dec[decit][j];
```

```
    CUDAmodule7 <<< nBlocks_n, blockSize >>> (m, n, r_d, a_d,
dec_d);
    cudaMemcpy(dec[decit], dec_d, sizeof(double)*n,
cudaMemcpyDeviceToHost);
    K=0; for(j=0;j<n;j++) K=K+dec[decit][j];
    for(j=0;j<n;j++) decsum[j]=decsum[j]+dec[decit][j];
    if((it>=convits)||(it>=maxits)){
        conv=1;
        for(j=0;j<n;j++)
if((decsum[j]!=0)&&(decsum[j]!=convits)) conv=0;
        if(((conv==1)&&(K>0))||(it==maxits)) dn=1;
    }
}

cudaMemcpy(a, a_d, sizeof(double)*m, cudaMemcpyDeviceToHost);
cudaMemcpy(r, r_d, sizeof(double)*m, cudaMemcpyDeviceToHost);

cudaFree(dec_d);
cudaFree(r_d);
cudaFree(k_d);
cudaFree(srp_d);
cudaFree(a_d);
cudaFree(s_d);
cudaFree(i_d);
cudaFree(mx1_d);
cudaFree(mx2_d);
```

Performance comparison is listed in the Table 2. For the sample data of 3,736 points, the AP program needs to complete 23,331 iterations to converge all features into 49 clusters. The serial C program needs about 6,459 seconds to generate the result, while the CUDA program needs about 615 seconds to complete the task, achieving a 10.5 speedup.

Result CPU-serial code	**Result of CUDA/GPU program**
 read data points Read similarities Read preferences m value: 13957696, n value: 3736 end of initialization : OK it# : 23331 Number of identified clusters: 49 Fitness (net similarity): -301.040077 Similarities of data points to exemplars: -124.711657 Preferences of selected exemplars: -176.328420 Number of iterations: 23331	22 January 2015 11:30:28 AM read data points Read similarities Read preferences m value: 13957696, n value: 3736 end of initialization : OK nBlocks_n:8, blockSize:512 it# : 23331 22 January 2015 11:40:43 AM Number of identified clusters: 49 Fitness (net similarity): -301.040077 Similarities of data points to exemplars: -124.711657 Preferences of selected exemplars: -176.328420

end of program real 107m39.286s user 107m33.663s sys 0m0.248s	Number of iterations: 23331 22 January 2015 11:40:43 AM end of program

Table 2. Performance comparison on AP serial C program vs. CUDA program on K20

5. Potential Extension and Broader Impacts

When the main frame of the AP algorithm could be transformed and implemented over a GPU, the same approach can be extended to transform the CUDA program to appropriate solutions doable on MIC. Since the majority of the AP program can be implemented by the embarrassingly parallel approach, the proposed solution has the potential to deploy more distributed computing processors (i.e. clusters of either GPUs or MICs) to achieve the goal of scalable AP computation over big geospatial datasets. Considering the broader impact of AP in the scientific community in general, the parallel version of AP on GPU/MIC and clusters of GPU/MIC will have broader impact in the future.

6. References

AFFINITY PROPAGATION FAQ. http://genes.toronto.edu/affinitypropagation/faq.html

Chehdi, K., Soltani, M., and Cariou, C. 2014. Pixel classification of large-size hyperspectral images by affinity propagation. Journal of Applied Remote Sensing. Vol. 8, Issue 1, 2014

Dueck, D. 2009. Affinity Propagation: Clustering Data by Passing Messages. Doctoral dissertation, University of Toronto.

Frey, B.J. and Dueck, D. 2007. Clustering by Passing Messages Between Data Points. Science 315, 972–976, February 2007

Frey, B.J. and Dueck, D. 2008. Response to Comment on "Clustering by Passing Messages between Data Points". Science 319, 726 (2008)

Guo, D. and Mennis, J. 2009. Spatial data mining and geographic knowledge discovery – An introduction. Computers, Environment and Urban Systems. Computers, Environment and Urban Systems, Vol. 33, No. 6. (November 2009), pp. 403-408.

Yang, C., Bruzzone, L., Sun, F., Lu, L., Guan, R. and Liang, Y. 2010. A Fuzzy-Statistics-Based Affinity Propagation Technique for Clustering in Multispectral Images. IEEE Transactions on Geoscience and Remote Sensing, Vol. 48, No. 6, June 2010, pp2647-2659.