Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Clustering Large Data Sets with Parallel Affinity Propagation

Robin Blasberg · Matthias K. Gobbert

Received: date / Accepted: date

Abstract Affinity propagation is a recently introduced clustering algorithm that involves iterative row and column updates of matrices. The method has great potential for large data sets, in particular if the number of clusters in the data set is also large and not known in advance. The key motivation for parallel computing is the large memory requirement for the matrices in the algorithm, and the formulation of the algorithm involving row and column oriented operations holds great potential for efficient parallelization. We present a memory-optimal implementation of the algorithm and show for a synthetic data set its excellent scalability on a distributed-memory cluster with high-performance interconnect network.

Keywords Pattern Recognition · Clustering · Affinity Propagation · Parallel Computing · Cluster Computing

1 Introduction

Affinity propagation is a clustering algorithm recently introduced by Frey and Dueck [1]. This iterative message exchange routine identifies a set of characteristic examples from the data points. Each data point is then associated with one of these exemplars. A cluster is the group of data points that has been linked with a particular exemplar. Frey and Dueck demonstrated diverse areas in which affinity propagation proved effective and obtained the solution faster than other clustering methods. Examples included facial image clustering, gene detection, and manuscript summarization [1]. The discussions in [2] and [3] resulting from the original introduction of the algorithm bring out that affinity propagation may be particularly useful if the data set consists of a large number of clusters. In this case, it is hard for the user to provide the correct

Robin Blasberg
Naval Research Laboratory, Washington, D.C.

Matthias K. Gobbert
Department of Mathematics and Statistics, University of Maryland, Baltimore County
Tel.: +1-410-455-2404
Fax: +1-410-455-1066
E-mail: gobbert@math.umbc.edu

number of clusters as input to the algorithm as is necessary for many other clustering algorithms. By contrast, the number of clusters emerges from the algorithm in affinity propagation based on the input of a so-called similarity matrix S and preference vector p. Specifically, for a data set of N data points, the components s_{ij} of the $N \times N$ similarity matrix $S = (s_{ij})$ represent the likeliness that point j is an exemplar for point i. If no other information is known, the similarity can be reasonably set to the negative squared distance between the two points. The N components p_i of the preference vector $p = (p_i)$ indicate the likelihood that a point i is an exemplar. If no a priori knowledge exists, the preference vector can be set to the mean or median similarity. The preference values affect the number of clusters that the affinity propagation code forms. That is, higher preference values tend to result in more clusters [1].

The affinity propagation algorithm involves two additional $N \times N$ matrices besides S. Each operation of the algorithm is either row or column oriented involving these matrices. This is an ideal algorithmic formulation for efficient parallelization of the method. Besides the obvious benefit of excellent speedup, the most important advantage of the parallel implementation is the pooling of all memory of several compute nodes that allows for the solution of much larger problems. This is important for affinity propagation, because its use of three $N \times N$ matrices makes it inherently memory intensive. For instance, to apply the algorithm to a data set with $N = 17{,}000$ data points requires at least 6.6 GB of memory. By pooling the memory of several nodes, it is possible to solve substantially larger cases such as with $N = 126{,}700$ data points which requires at least 367 GB total memory.

Memory limitations are a pervasive problem when working with large data sets, and a parallel solution is not uncommon. For instance, an examination of many data clustering algorithms shows an underlying parallel foundation [4]. Parallel implementations of k-means exist [5], and associated performance studies have been reported [6]. Since k-means involves only vectors of length N (instead of matrices of size $N \times N$), it is possible to solve much larger problems, potentially with millions of data points, than with affinity propagation. However, k-means requires the user to provide the number of clusters as input, which is problematic particularly for large numbers of clusters.

Another approach to solving large problems is a sparse version of affinity propagation [1]. This approach will work for problems where it is known a priori for a large number of pairs of data points that they do *not* belong to the same cluster. But for general clustering problems, for which sparse affinity propagation cannot be used and that have an unknown, potentially large, number of clusters, the parallel implementation of (non-sparse) affinity propagation is vital to attack the problem. This is particularly necessary, because problems with large numbers of clusters will typically also involve large numbers of data points.

In the following Section 2, we describe our parallel implementation of affinity propagation in more detail. In order to test the algorithm on data sets of any desired size with a known solution, we have designed a synthetic data set. Section 3 details its design and lists the results of validation studies. Section 4 presents the results of both serial and parallel performance studies for our code that demonstrate its excellent scalability and ability to solve large clustering problems. Finally, Section 5 summarizes our conclusions.

2 Parallel Affinity Propagation

Affinity propagation functions by identifying similar data points in an iterative process [1]. The data set is given as N data points, and the goal of the algorithm is to cluster groups of data points that are close to each other. The method of affinity propagation is based on a criterion embedded in a similarity matrix $S = (s_{ij})$ where each component S_{ij} quantifies the closeness between data points i and j. We follow the default suggested in [1] by using the negative square of the Euclidean distance between data points.

The algorithm updates a matrix A of 'availabilities' and a matrix R of 'responsibilities' iteratively until the computed clusters do not change for `convits` many iterations. Our memory-optimal code uses the three matrices S, A, and R as the only variables of significant size, namely $N \times N$ double-precision numbers. To distribute the memory across all parallel processes, all matrices are split consistently across the p parallel processes by groups of adjacent columns.

Our implementation assumes that the similarity matrix is symmetric. That is, it is assumed that point j has the same likelihood of being an exemplar for point i as point i's likelihood of being an exemplar for point j. As many quantities as possible are computed for S by using only information that is local to each parallel process. This minimizes the number of parallel communications. As a result of our design, we need only two communication commands in each iteration, both of which are `MPI_Allreduce` commands from the MPI library for parallel communications. Other MPI commands appear in the initialization and the post-processing, but only the commands inside the iteration loop impact the performance significantly. Our code is written in C, and we use the MVAPICH2 implementation of MPI.

Our code sets all preference values to the mean of the similarity matrix. This differs from the original [1] which uses the median of the similarity values as the default for the preference vector. Serial validation tests confirm that this choice does not impact the clustering significantly.

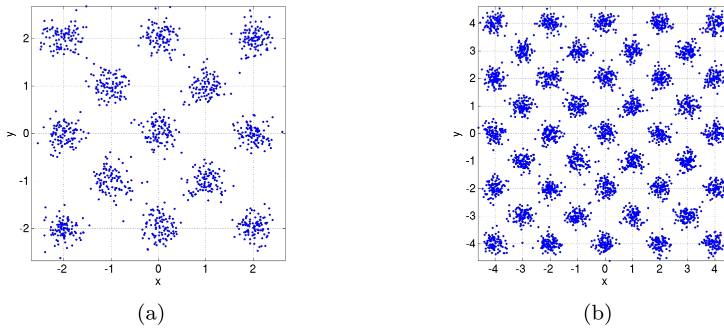
Since affinity propagation is based on matrix calculations, it has relatively large memory requirements. For instance, a data set with $N = 126,700$ data points requires 367,421 MB or over 367 GB. This kind of memory requirement cannot be accommodated on a serial computer but requires the combined memory of many nodes of a parallel computer.

In the development of our code, we used the Matlab version of the Frey and Dueck affinity propagation code available at `www.psi.toronto.edu/affinitypropagation`. Starting from this original Matlab code, we developed a simplified version that both avoids all unnecessary variables and translates the high-level matrix and vector operations in Matlab to for loops. Our simplified Matlab code was then rewritten in C with the major matrix operations being broken out such that columnar manipulation would be possible. This enabled efficient parallelization of the C code.

3 Data Set and Validation

3.1 Synthetic Test Data Set

The focus of this work is to analyze the size of problems that can be solved by a parallel implementation of affinity propagation and to study the scalability of this solution to many parallel processes. Therefore, we need on the one hand a test data set for which



(a) (b)

Fig. 1 Examples of synthetic data sets (a) with $N = 1{,}300$ points in $q = 13$ classes and (b) with $N = 4{,}100$ points in $q = 41$ classes

we can readily determine that the problem is solved correctly and that this solution is not affected significantly by the number of parallel processes used. Thus, we design a synthetic data set of arbitrary size N with known true clusters, because this allows the scalar measures of entropy and purity [6, 7] to be used to judge the correctness of the computed results. On the other hand, we want to choose the size N of the data set as any desired value to analyze memory usage and scalability over the full range possible. To this end, the use of a synthetic data set is appropriate, because both memory usage and parallel scalability only depend on the size of the problem, not on the data themselves.

Our synthetic data set consists of N data points (x_i, y_i) in q true clusters called classes such that equal numbers of N/q data points are in each class. For chosen input values N and q, the data set is created by scattering N/q points about one centroid computed for each of the q classes. Specifically, inspired by geographic applications which have clusters of data spread out across the plane, the centroids are first distributed in a checkerboard fashion about the origin of the (x, y)-plane at points with integer coordinates $(x, y) = (i, j)$, for which $i + j$ is an even number. This design results in a uniform spacing between neighboring centroids. The N/q points in each class are then placed by scattering their x and y coordinates about their centroid with a normal distribution with standard deviation $\sigma = 0.2$. This deviation is significantly smaller than the distance between neighboring centroids, hence nearly all data points are indeed closest to the centroid used in their construction. Two examples of data sets computed in this way are shown in Figure 1.

Table 1 (a) shows the parameters for all data sets considered in the following studies. The values of N range from 500 to 126,700 along with q classes from 5 to 181. The values of N are roughly equal to powers of 2 which is chosen so that the memory usage quadruples approximately for each increase in N. The controlled approach used to create the data set allows the evaluation of the quality of a generated data set. Specifically, we can determine whether a data point is closer to another centroid than the one used in its construction. This is the definition for a 'bad' point as counted in the column with label `numbadpts`. In turn, the column `Pdata` shows the fraction of 'good' data points as percent of the total number of data points. The numbers for `Pdata` indicate that all data sets have relatively well separated clusters with very few points closer to the 'wrong' than the 'right' centroid. Thus, the clustering algorithm

Table 1 (a) Properties and (b) results for all data sets considered

(a) Properties N	q	`numbadpts`	`Pdata`(%)	`mean`
500	5	0	100.00	−3.38
1,300	13	0	100.00	−8.82
2,500	25	1	99.96	−16.78
4,100	41	6	99.85	−27.50
9,150	61	8	99.91	−40.82
17,000	85	18	99.89	−56.85
33,900	113	27	99.92	−75.56
65,250	145	56	99.91	−96.84
126,700	181	97	99.92	−120.87
(b) Results N	`its`	k	entropy	purity
500	153	5	0.0070	0.9980
1,300	131	13	0.0017	0.9992
2,500	130	25	0.0043	0.9972
4,100	131	41	0.0031	0.9978
9,150	135	61	0.0074	0.9931
17,000	133	85	0.0068	0.9912
33,900	139	113	0.0087	0.9890
65,250	156	145	0.0121	0.9828
126,700	172	181	0.0194	0.9672

should be able to produce a correct result, meaning that data points are associated with the correct cluster. For affinity propagation, this also includes the expectation that the clustering algorithm determine the correct number of clusters automatically. In our simulations, we follow the default suggested in [1] by using the negative square of the Euclidean distance between the data points as definition for the components of the similarity matrix. We modified the computation of the preference vector from the original median to using the mean, because this allowed for better parallelization and is not expected to affect the results significantly. The final column of Table 1 (a) lists the preference mean for each data set that is used in the input to affinity propagation.

3.2 Validation Results

The focus of this work is on parallel scalability and memory usage of our parallel code. In validation studies, we use the concepts of entropy and purity as convenient scalars to measure the quality of the clustering result produced. These quantities measure the quality relative to the classes in the data set. That is why it was important to use data sets for which the true clusters (i.e., classes) have very few data points that are closer to other centroids.

The term entropy is used to describe the heterogeneity of the clusters. Mathematically, the entropy of a particular cluster S_r is defined as [6, 7]

$$E(S_r) = -\frac{1}{\log q} \sum_{i=1}^{q} \frac{n_r^i}{n_r} \log \frac{n_r^i}{n_r},$$

where q is the number of classes, n_r is the size of the cluster S_r, and n_r^i is the number of points from the ith class that were put in the rth cluster. The entropy of the entire clustering result is then a weighted sum of all the individual entropies as given by

$$Entropy = \sum_{r=1}^{k} \frac{n_r}{N} E(S_r).$$

As can be noted from this equation, the ideal value for entropy is 0. Entropy increases as it moves further away from this ideal value.

In contrast to entropy, purity is used to describe the homogeneity of the clusters. Mathematically, the purity of a cluster S_r is defined as [6,7]

$$P(S_r) = \frac{1}{n_r} \max_i (n_r^i).$$

The purity of the entire clustering result is then a weighted sum of all of the individual clustering purities as given by

$$Purity = \sum_{r=1}^{k} \frac{n_r}{N} P(S_r)$$

The ideal value for purity is 1. Purity values decrease towards 0 as the purity moves further away from this ideal value.

Entropy and purity are essentially opposites with entropy describing the degree of disorder in the cluster while purity describes the degree of order in a cluster. To further illustrate the difference between entropy and purity, consider the following simple example: Suppose two clusters are formed with cluster 1 being the set $\{A, A, B, B\}$ and cluster 2 the set $\{A, A, B, C\}$, where the letter identifies the class of the data point. It can be seen that both cluster 1 and cluster 2 have the same purity since both of these clusters have at most two elements from a single class. However, cluster 1 will have a lower (i.e., better) entropy value than cluster 2, because cluster 1 contains elements from a smaller number of different classes than cluster 2. That is, cluster 1 only has elements from classes A and B as opposed to cluster 2 which has elements from classes A, B, and C.

To examine whether our code could solve all problems correctly, we performed validation studies for the data sets specified in Table 1 (a). Table 1 (b) shows the results of affinity propagation applied to this data by listing the number of iterations `its`, the number k of clusters formed by the algorithm, as well as the entropy and purity measures for each data set. We note first that the algorithm finds automatically the correct number of clusters k, whose values agree exactly with the value q of classes (i.e., true clusters) in the data sets shown in Table 1 (a). Moreover, the near-zero entropy values and the purity values near 1 demonstrate that affinity propagation clustered even large data sets successfully. It is also interesting to observe that the number of iterations does not grow with the problem size. This makes this algorithm attractive for large data sets.

The results in Table 1 (b) were obtained using numerical parameters set to default values suggested by Frey and Dueck in their Matlab implementation of the algorithm. Specifically, we used `maxits` = 1,000 for the maximum number of iterations allowed, `convits` = 100 for the number of iterations over which the convergence test is applied, and the damping parameter $\lambda = 0.9$. The results in Table 1 (b) are taken from the

runs with $p = 128$ parallel processes. They were carefully checked against all other runs, including serial and both original and our Matlab codes (for data sets fitting in serial memory), and found in agreement in all cases which confirms the correctness of all results. Specifically, runs with different numbers of parallel processes resulted in entropy and purity values with deviations within reasonable multiples of round-off, indicating that a few points were sometimes clustered differently. This is expected as the true data sets do have a few points that are closer to other centroids than their own class's for the larger values of N, as seen in Table 1 (a). But in all cases, the number k of clusters found automatically by affinity propagation as well as the number of iterations `its` were exactly the same for the various numbers of parallel processes tested. This latter fact is important to ensure that the parallel code indeed does the same amount of work for a fixed problem size N, independent of the number of processes p.

4 Performance Results

The code was run on the cluster `hpc` in the UMBC High Performance Computing Facility (HPCF; `www.umbc.edu/hpcf`). This distributed-memory cluster consists (presently) of 32 compute nodes, integrated by IBM. Each of these nodes is equipped with two dual-core AMD Opteron processors and 13 GB of memory, for a total system memory of 416 GB. The compute nodes are connected by a high-performance InfiniBand interconnect network. We use the MVAPICH2 implementation of MPI. An additional head node for compiling and job submission has the same processors as the compute nodes, but an extended 17 GB of memory.

4.1 Serial Performance

Since our code development started with the original Matlab code by Frey and Dueck and involved our own intermediate Matlab code in the development of our parallel C code, it makes sense to compare their memory usage and performance with each other in serial. This gives a feel for the magnitude of the problem and will motivate the use of parallel computing. The first columns of Table 2 (a) show the predicted memory usage in MB for all data sets listed in Table 1. The memory prediction for 1 matrix, stored in double-precision with 8 bytes per number, shows how the memory requirement grows rapidly with N. Recall that the algorithm uses at a minimum 3 matrices, namely the matrix A of 'availabilities', the matrix R of 'responsibilities', as well as the similarity matrix S. The remaining columns in Table 2 (a) collect the memory usage observed during the serial run of each code for those cases that could be accommodated in serial; to accommodate the original Matlab code for the data set with $N = 17{,}000$, the serial runs were performed on the head node of the cluster which has the same processors as the compute nodes, but an extended 17 GB of memory compared to 13 GB on the compute nodes. The original Matlab code uses clearly some additional large matrices. Although our Matlab memory requirements are noticeably less than those of the original Matlab code, both columns of Matlab results underscore the additional memory requirements of running Matlab software. By contrast, it can be seen from the memory observation for our C code that the amount of memory we need is far closer to the predicted memory estimates than the original Matlab code. The results in Table 2 (a) also show that the difference between the serial memory

Table 2 Memory usage and performance of the original Matlab code, our Matlab code, and our C code in serial. For the small $N = 500$ case, N/A indicates that the run finished too fast to observe memory usage.

N	predicted memory for 1 $N \times N$ matrix	predicted memory for 3 matrices (S, A, R)	observed memory for original Matlab	observed memory for our Matlab	observed memory for our C code
500	2	6	825	800	N/A
1,300	13	39	937	845	100
2,500	48	143	1,161	999	202
4,100	128	385	1,661	1,334	444
9,150	639	1,916	4,650	3,376	1,978
17,000	2,205	6,615	$\approx$ 13,800	9,625	6,680
33,900	8,768	26,303	N/A	N/A	N/A
65,250	32,483	97,448	N/A	N/A	N/A
126,700	122,474	367,421	N/A	N/A	N/A

(a) Predicted and observed memory usage in MB in serial

N	original Matlab		our Matlab code		our C code	
	initialization	clustering	initialization	clustering	initialization	clustering
500	1.67	4.39	0.49	12.46	0.05	1.71
1,300	11.20	20.34	2.85	68.61	0.18	9.57
2,500	41.14	70.58	10.73	249.76	0.80	35.29
4,100	112.44	192.69	28.59	656.66	1.46	95.99
9,150	581.67	1066.52	144.47	3345.16	7.26	498.37
17,000	1447.24	3676.90	388.16	11780.80	22.77	1798.04

(b) Observed wall clock times in seconds in serial

requirements of the original Matlab code and the serial memory requirements of our C code grows increasingly wider as the data set size grows larger. This would indicate that, even in its serial form, our C code holds a noticeable capacity advantage over both Matlab codes.

Table 2 (b) shows the observed wall clock times in seconds for the same runs as Table 2 (a). The time taken for the initialization of the similarity matrix and the preference vector is reported under the heading initialization. The time for the affinity propagation algorithm is shown separately under the heading clustering. This segregation brings out the potentially significant fraction of time needed for the initialization in Matlab, while it is fairly insignificant in C. Thus, the separation allows for a clearer comparison of the implementations. Our Matlab code was used as an intermediate step for the development of the C code. That is, we used it to reduce the number of variables to the memory-optimal case of 3 matrices, but we also replaced various Matlab functions by explicit code involving, e.g., for loops, for easy translation of the code to C. Therefore, we do not expect our Matlab code to perform well, and this is exhibited by the timing results for the clustering. Interestingly though, our times for the initialization are faster than the original Matlab, while still much slower than the C code. Specifically, a look at the clock times of the serial runs for various N values shows that the time that it takes to create the similarity matrix is roughly 3 times faster with our Matlab initialization code. The speed improvement associated with our C code

for the initialization function is even more pronounced. The clock times for initializing the similarity matrices using our C code takes less than 2% of the initialization time required by the original Matlab code. Additionally, the clock times of the serial runs also show that the speed of the clustering routine is at least doubled with our C code.

4.2 Parallel Performance

The serial memory usage and run times for the larger data sets in Table 2 bring out the two key motivations for parallel computing: Combining the memory of several nodes allows for the solution of significantly larger problems, and the run times for a problem of a given, fixed size can be potentially dramatically reduced by spreading the work across a group of parallel processes. Both of these advantages are brought out by this application in the following.

Since affinity propagation is based on matrix calculations, it has relatively large memory requirements. This can be seen concretely in Table 3 (a) which shows in the column $p = 1$ the total memory requirements in MB for the three $N \times N$ matrices using 8 bytes per double-precision matrix component. The remaining columns list the memory requirement for each parallel process if the three matrices are split into p equally large portions across the processes. For instance, a data set with $N = 126{,}700$ data points requires 367,421 MB or over 367 GB. This kind of memory requirement cannot be accommodated on a serial computer but requires the combined memory of many nodes of a parallel computer. Specifically, on `hpc` with 13 GB of memory per

Table 3 Memory usage on `hpc` using MVAPICH2 in MB per process. For small N values, N/A indicates that the run finished too fast to observe memory usage. For large N values, N/A indicates that the case required more memory than available per node.

(a) Predicted memory usage in MB per process

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	6	3	1	1	< 1	< 1	< 1	< 1
1,300	39	19	10	5	2	1	1	< 1
2,500	143	72	36	18	9	4	2	1
4,100	385	192	96	48	24	12	6	3
9,150	1,916	958	479	240	120	60	30	15
17,000	6,615	3,307	1,654	827	413	207	103	52
33,900	26,303	13,152	6,576	3,288	1,644	822	411	205
65,250	97,448	48,724	24,362	12,181	6,090	3,045	1,523	761
126,700	367,421	183,711	91,855	45,928	22,964	11,482	5,741	2,870

(b) Observed memory usage on `hpc` using MVAPICH2

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1,300	100	83	76	72	N/A	N/A	N/A	N/A
2,500	202	133	100	86	80	72	74	N/A
4,100	444	253	160	113	93	83	80	75
9,150	1,978	1,020	543	305	189	128	102	87
17,000	6,680	3,372	1,719	894	483	276	176	123
33,900	N/A	N/A	6,641	3,355	1,715	893	485	277
65,250	N/A	N/A	N/A	N/A	6,163	3,117	1,599	835
126,700	N/A	N/A	N/A	N/A	N/A	$\approx 11{,}600$	5,819	2,949

compute node, we need all 32 nodes to accommodate a data set of this size. Table 3 (b) shows the memory usage observed for our code. We observe that the memory required in reality is more than predicted but within reason when smaller variables and loading of required libraries are taken into consideration.

The ideal behavior of parallel code for a fixed problem size using p parallel processes is that it be p times as fast as with 1 process. If $T_p(N)$ denotes the wall clock time for a problem of a fixed size parametrized by the number N using p processes, then the quantity $S_p := T_1(N)/T_p(N)$ measures the *speedup* of the code from 1 to p processes, whose optimal value is $S_p = p$. The *efficiency* $E_p := S_p/p$ characterizes in relative terms how close a run with p parallel processes is to this optimal value, for which $E_p = 1$. This behavior described here for speedup and efficiency for a given, fixed problem size is known as strong scalability of parallel code.

Table 4 lists the results of a performance study for strong scalability. Each row lists the results for one problem size parametrized by the number of data points N. Each column corresponds to the number of parallel processes p used in the run. The runs for Table 4 cluster these p parallel MPI processes on the compute nodes as tightly as possible by using all 4 cores (both cores on both processors) on each of the $p/4$ nodes used for the job. For the runs with $p = 1$ and $p = 2$, the job uses only one node, with several cores idling, so that the job can use the entire memory of the node for the job. Additionally, the table reports data marked with an asterisk that extends the table as much as possible. These are cases, where the memory of a node could not accommodate the use of all 4 cores. For instance, for $N = 33{,}900$, the case $p = 8$ requires about 3.4 GB of memory per process according to Table 3 (b). This cannot be accommodated by clustering 4 processes per node. Thus, the result of clustering only 2 processes per node is reported for this case. Analogously, the case $p = 4$ for $N = 33{,}900$ required over 6.6 GB per process, which can only be accommodated by spreading the run over 4 nodes. Tables with complete results for the configurations with 1 and 2 processes per node, respectively, are available in [8], which also contains a comparison to results using the OpenMPI implementation of MPI.

Comparing adjacent columns in the raw timing data in Table 4 (a) indicates that using twice as many processes speeds up the code by nearly a factor of two for nearly all cases, except the smallest ones and the case $N = 126{,}700$; see below. To quantify this more clearly, the observed speedup in Table 4 (b) is computed, which shows near-optimal with $S_p \approx p$ for all cases of N up to $p = 32$, except the smallest cases, and still excellent results beyond that for the larger data sets, except for $N = 126{,}700$. For the data sets that are too large for serial runs, the speedup is re-defined to start with the first available value of p. For instance, for $N = 33{,}900$, we compute $S_p := 4T_4/T_p$, since $p = 4$ is the first available case; analogous re-definitions are used for $N = 65{,}250$ and 126,700. These re-definitions use the data marked with an asterisk that was obtained with several cores idling on the node, which has the potential for performing much better. Thus, this approach of analyzing the data might *underestimate* the performance achieved by our code and is conservative in nature, but it makes the most amount of data available in the table using a consistent scale to allow for comparisons. Finally, Table 4 (c) lists the observed efficiency $E_p = S_p/p$. This clearly demonstrates the excellent scalability of the code all the way to $p = 128$ for the larger data sets. Notice that we are comparing to the case of $p = 1$, which was run on a dedicated node with 3 cores idling. This often results in severe degradation of performance compared to $p = 2$ and $p = 4$. It is a good indicator of both MPI and hardware performance that this is not the case here. We note that the apparent loss of speedup and efficiency for

Table 4 Performance on `hpc` using MVAPICH2 by number of processes used with 4 processes per node except for $p = 1$ which uses 1 process per node and $p = 2$ which uses 2 processes per node. N/A indicates that the case required more memory than available. Data marked by an asterisk indicates that there was insufficient memory to run 4 processes per node, but data was obtained with 1 or 2 processes per node in order to extend the comparisons in the table as much as possible.

(a) Wall clock time in seconds

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
500	1.71	0.91	0.53	0.22	0.14	0.08	0.06	0.05
1,300	9.57	5.05	2.64	1.32	0.69	0.36	0.18	0.12
2,500	35.29	18.48	9.20	4.77	2.39	1.24	0.67	0.42
4,100	95.99	49.72	25.11	12.78	6.43	3.31	1.74	1.12
9,150	498.37	260.85	130.42	64.99	32.86	16.78	8.53	4.64
17,000	1798.04	932.81	440.05	223.39	112.05	56.69	28.91	15.43
33,900	N/A	N/A	*2287.98	*1223.54	568.01	300.07	143.58	78.81
65,250	N/A	N/A	N/A	N/A	*2593.53	*1595.91	718.26	402.58
126,700	N/A	N/A	N/A	N/A	N/A	*6160.75	*4493.82	2720.27

(b) Observed speedup S_p

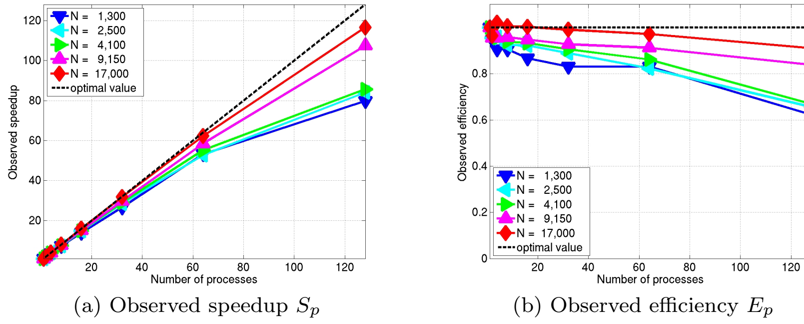
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
500	1.0000	1.8791	3.2264	7.7727	12.2143	21.3750	28.5000	34.2000
1,300	1.0000	1.8950	3.6250	7.2500	13.8696	26.5833	53.1667	79.7500
2,500	1.0000	1.9096	3.8359	7.3983	14.7657	28.4597	52.6716	84.0238
4,100	1.0000	1.9306	3.8228	7.5110	14.9285	29.0000	55.1667	85.7054
9,150	1.0000	1.9106	3.8213	7.6684	15.1665	29.7002	58.4256	107.4073
17,000	1.0000	1.9276	4.0860	8.0489	16.0468	31.7171	62.1944	116.5288
33,900	N/A	N/A	*4.0000	*7.4799	16.1123	30.4993	63.7409	116.1264
65,250	N/A	N/A	N/A	N/A	*16.0000	*26.0018	57.7736	103.0764
126,700	N/A	N/A	N/A	N/A	N/A	*32.0000	*43.8700	72.4722

(c) Observed efficiency E_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
500	1.0000	0.9396	0.8066	0.9716	0.7634	0.6680	0.4453	0.2672
1,300	1.0000	0.9475	0.9062	0.9062	0.8668	0.8307	0.8307	0.6230
2,500	1.0000	0.9548	0.9590	0.9248	0.9229	0.8894	0.8230	0.6564
4,100	1.0000	0.9653	0.9557	0.9389	0.9330	0.9062	0.8620	0.6696
9,150	1.0000	0.9553	0.9553	0.9586	0.9479	0.9281	0.9129	0.8391
17,000	1.0000	0.9638	1.0215	1.0061	1.0029	0.9912	0.9718	0.9104
33,900	N/A	N/A	*1.0000	*0.9350	1.0070	0.9531	0.9960	0.9072
65,250	N/A	N/A	N/A	N/A	*1.0000	*0.8126	0.9027	0.8053
126,700	N/A	N/A	N/A	N/A	N/A	*1.0000	*0.6855	0.5662

the largest case of $N = 126{,}700$ is a consequence of the increased effectiveness possible when only using 1 core per node, as in the case $p = 32$ that it is compared to.

The customary visualizations of speedup and efficiency are presented in Figure 2 (a) and (b), respectively, for five intermediate values of N that still allow for serial runs, thus the canonical definition of speedup. Figure 2 (a) shows very clearly the excellent speedup all the way up to $p = 64$ parallel processes for all cases shown and excellent performance for the two larger cases all the way to $p = 128$. The efficiency plotted in Figure 2 (b) is directly derived from the speedup and confirms these conclusions. An efficiency plot is often useful because it can better bring out interesting features for small values of p that are hard to tell in a speedup plot. Here, we notice that some variability of the results for small p is visible. However, there is not a large drop in efficiency from



(a) Observed speedup S_p (b) Observed efficiency E_p

Fig. 2 Performance on `hpc` using MVAPICH2 by number of processes used with 4 processes per node except for $p = 1$ which uses 1 process per node and $p = 2$ which uses 2 processes per node.

$p = 1$ to $p = 2$ and $p = 4$, as we already noted in the discussion on Table 4 (c). This is an excellent result, because the runs with $p = 1$ and $p = 2$ involve some idling cores on the node used, thus have added potential for performance, compared to the runs using all four cores per node. It is customary in results of strong scalability studies for given, fixed problem sizes that the speedup is better for larger problems since the increased communication time for more parallel processes does not dominate over the calculation time as quickly as it does for small problems. Thus, the progression in speedup performance from smaller to larger data sets seen in Figures 2 (a) and (b) is expected. To see this most clearly, it is vital to have the precise data shown in Tables 4 (b) and (c) available and not just their graphical representation in Figure 2.

The results presented so far were for runs using all 4 cores per node (except for $p = 1$ and $p = 2$ processes or unless memory requirements prevented this). Table 5 summarizes the observed wall clock times for each of the data sets considered and contrasts the use of 1, 2, or 4 processes per node. Specifically, the upper-left entry of each sub-table with 1 process per node on 1 node represents the serial run of the code, while the lower-right entry of each sub-table lists the time for running 4 processes on all 32 nodes using both cores of both dual-core processors on each node for a total of 128 parallel processes. This table is designed to analyze the advantage of using all 4 cores per node when using a given number of nodes for the job. It is apparent that, with the exception of the largest numbers of nodes for the smallest data set, the execution time of each problem is, in fact, vastly reduced with doubling the numbers of processes per node, albeit not quite halved. These results confirm that it is not just effective to use both processors on each node, but it is also effective to use both cores of each dual-core processor simultaneously. This is clearly also true for the case of $N = 126{,}700$ data points, despite the fact that the data for this case in Table 4 shows that the speed improvement associated with doubling the number of cores used is less than optimal.

Table 5 Performance on `hpc` using MVAPICH2. Wall clock time in seconds for the solution of problems with N data points using 1, 2, 4, 8, 16, 32 compute nodes with 1, 2, and 4 processes per node. N/A indicates that the case required more memory than available.

(a) $N = 500$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	1.71	0.87	0.45	0.22	0.12	0.08
2 processes per node	0.91	0.47	0.22	0.13	0.08	0.06
4 processes per node	0.53	0.22	0.14	0.08	0.06	0.05
(b) $N = 1{,}300$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	9.57	4.77	2.45	1.23	0.67	0.34
2 processes per node	5.05	2.58	1.31	0.68	0.36	0.18
4 processes per node	2.64	1.32	0.69	0.36	0.18	0.12
(c) $N = 2{,}500$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	35.29	17.58	8.96	4.48	2.28	1.20
2 processes per node	18.48	9.25	4.69	2.38	1.32	0.68
4 processes per node	9.20	4.77	2.39	1.24	0.67	0.42
(d) $N = 4{,}100$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	95.99	47.83	23.98	12.14	6.13	3.18
2 processes per node	49.72	25.06	12.62	6.41	3.37	1.75
4 processes per node	25.11	12.78	6.43	3.31	1.74	1.12
(e) $N = 9{,}150$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	498.37	251.72	126.76	63.16	31.81	16.62
2 processes per node	260.85	130.74	65.31	32.83	17.05	8.70
4 processes per node	130.42	64.99	32.86	16.78	8.53	4.64
(f) $N = 17{,}000$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	1798.04	864.00	424.48	214.80	109.19	56.27
2 processes per node	932.81	448.18	220.77	111.99	58.64	30.32
4 processes per node	440.05	223.39	112.05	56.69	28.91	15.43
(g) $N = 33{,}900$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	N/A	N/A	2287.98	1004.23	492.52	255.92
2 processes per node	N/A	N/A	1223.54	553.48	287.41	153.58
4 processes per node	N/A	N/A	568.01	300.07	143.58	78.81
(h) $N = 65{,}250$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	N/A	N/A	N/A	N/A	2593.53	1291.01
2 processes per node	N/A	N/A	N/A	N/A	1595.91	742.32
4 processes per node	N/A	N/A	N/A	N/A	718.26	402.58
(i) $N = 126{,}700$	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**32 nodes**
1 process per node	N/A	N/A	N/A	N/A	N/A	6160.75
2 processes per node	N/A	N/A	N/A	N/A	N/A	4493.82
4 processes per node	N/A	N/A	N/A	N/A	N/A	2720.27

5 Conclusions

Affinity propagation is a recent clustering method whose algorithm involves iterative updates of the rows and columns of three large matrices. The method has great potential for data sets consisting of large but unknown numbers of clusters. Problems of this type will typically also involve large numbers of data points. We observe that the number of iterations does not grow with the problem size, adding to the attractiveness of the method. However, to accommodate large data sets, the memory needed for the three (non-sparse) matrices is a challenge. Parallel computing that pools the memory of several compute nodes allows us to solve these large problems. Table 4 demonstrates that an efficient parallel implementation in MPI is possible and solves these problems with excellent scalability on modern compute nodes with a state-of-the-art interconnect network. Table 5 analyzes the use of different numbers of cores of the two dual-core processors on each compute node and clearly shows the advantage of using all cores available. All these results make a parallel implementation of affinity propagation an attractive tool for clustering large data sets with large but unknown numbers of clusters.

Acknowledgements The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources.

References

1. Frey, B.J., Dueck, D.: Clustering by passing messages between data points. Science **315**, 972–976 (2007)
2. Brusco, M.J., Köhn, H.F.: Comment on “Clustering by passing messages between data points”. Science **319**, 726c (2008)
3. Frey, B.J., Dueck, D.: Response to comment on “Clustering by passing messages between data points”. Science **319**, 726d (2008)
4. Zhang, B., Hsu, M.: Scale up center-based data clustering algorithms by parallelism. Tech. Rep. HPL–2000–6, Hewlett-Packard Company (2000)
5. Pataneè, G., Russo, M.: Parallel clustering on a commodity supercomputer. In: Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, vol. 3, pp. 575–580 (2000)
6. Xu, S., Zhang, J.: A parallel hybrid web document clustering algorithm and its performance study. J. Supercomput. **30**, 117–131 (2004)
7. Zhao, Y., Karypis, G.: Criterion functions for document clustering experiments and analysis. Tech. rep., Department of Computer Science/Army HPC Research Center, University of Minnesota (2002)
8. Blasberg, R., Gobbert, M.K.: MVAPICH2 vs. OpenMPI for a clustering algorithm. Tech. Rep. HPCF–2008–7, UMBC High Performance Computing Facility, University of Maryland, Baltimore County (2008). URL www.umbc.edu/hpcf