Parallel Performance Studies for a Clustering Algorithm

Robin V. Blasberg* and Matthias K. Gobbert†

*Naval Research Laboratory, Washington, D.C.

†Department of Mathematics and Statistics, University of Maryland, Baltimore County, gobbert@math.umbc.edu

Abstract

Affinity propagation is a clustering algorithm that functions by identifying similar datapoints in an iterative process. Its structure allows for taking full advantage of parallel computing by enabling the solution of larger problems and by solving them faster than possible in serial. We show that our memory-optimal implementation with minimal number of communication commands per iteration performs excellently on the distributed-memory cluster hpc and that it is efficient to use all 128 processor cores currently available.

1 Introduction

Affinity propagation is a relatively new clustering algorithm introduced by Frey and Dueck [2] that functions by identifying similar datapoints in an iterative process. A key advantage of the algorithm is that it does not require the user to predetermine the number of clusters and is thus useful particularly in the case of large numbers of clusters in the data. If N denotes the number of datapoints in the given dataset, a memory-optimal implementation of the algorithm requires three $N \times N$ matrices. The memory requirements for the algorithm grow very rapidly with N, for instance, a dataset with $N = 17{,}000$ datapoints needs about 6.6 GB of memory. However, the systematic structure of the algorithm allows for its efficient parallelization with only two parallel communication commands in each iteration. Thus, both the larger memory available and the faster run times achievable by using several nodes of a parallel computer demonstrate the combined advantages of a parallel implementation of the algorithm. The structure of the algorithm and its memory requirements are discussed in more detail in Section 2. Due to its excellent potential for scalability, the algorithm is also an ideal candidate for evaluating the hardware of a parallel cluster in extension of earlier studies such as [3].

The distributed-memory cluster hpc in the UMBC High Performance Computing Facility (HPCF, `www.umbc.edu/hpcf`) has an InfiniBand interconnect network and 32 compute nodes each with two dual-core processors (AMD Opteron 2.6 GHz with 1024 kB cache per core) and 13 GB of memory per node for a total of up to four parallel processes to be run simultaneously per node. This means that up to 128 parallel MPI processes can be run and the cluster has a total system memory of 416 GB. Section 3 describes the parallel scalability results in detail and provides the underlying data for the following summary results. Table 1 summarizes the key results of the present study by giving the wall clock time (total time to execute the code) in seconds. We consider nine progressively larger datasets, as indicated by the number of datapoints N, resulting in problems with progressively larger memory requirements. The parallel implementation of the numerical method is run on different numbers of nodes from 1 to 32 with different numbers of processes per node used. Specifically, the upper-left entry of each sub-table with 1 process per node on 1 node represents the serial run of the code, which takes 1.73 seconds for the dataset with $N = 500$ datapoints. 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 working together to solve the problem which takes 0.13 seconds for the $N = 500$ case. More strikingly, one realizes the advantage of parallel computing for a case with $N = 17{,}000$ datapoints requiring 6.6 GB of memory: The serial run of about 29¼ minutes (1754.29 seconds) can be reduced to about 24 seconds using 128 parallel processes. Yet, the true advantage of parallelizing the algorithm is evident for a problem with $N = 126{,}700$ datapoints that uses over 367 GB of memory. To solve this problem requires the combined memory of all 32 nodes of the cluster in order to be solved at all. Thus, a parallel implementation allows the solution of a problem that simply could not be solved before in serial, and it moreover takes only the very reasonable amount of 44 minutes (2641.05 seconds) to complete.

The results in Table 1 are arranged to study two key questions: (i) "Does the code scale optimally to all 32 nodes?" and (ii) "Is it worthwhile to use multiple processors and cores on each node?" The first question addresses the quality of the throughput of the InfiniBand interconnect network. The second question sheds light on the quality of the architecture within the nodes and cores of each processor.

(i) Reading along each row of Table 1, the wall clock time approximately halves as the number of nodes used doubles for all cases of N except for large numbers of nodes for the smallest datasets. That is, by being essentially proportional to the number of nodes used, the speedup is nearly optimal for all cases of significant size which are the cases for which parallel computing is relevant. This is discussed in more detail in Section 3 in terms of the number of parallel processes.

(ii) To analyze the effect of running 1, 2, or 4 parallel processes per node, we compare the results column-wise in each sub-table. It is apparent that, with the exception of the largest numbers of nodes for the smallest dataset, 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 are still excellent and confirm that it is not just effective to use both processors on each node, but also to use both cores of each dual-core processor simultaneously. Roughly, this shows that the architecture of the IBM nodes purchased in 2008 has sufficient capacity in all vital components to avoid creating any bottlenecks in accessing the memory of the node that is shared by the processes. These results thus justify the purchase of compute nodes with two processors (as opposed to one processor) and of dual-core processors (as opposed to single-core processors).

Table 1: Performance on hpc using OpenMPI. 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	1.73	0.91	0.44	0.22	0.13	0.08
2 processes per node	0.92	0.46	0.21	0.12	0.09	0.07
4 processes per node	0.45	0.22	0.13	0.10	0.11	0.13

(b) $N = 1{,}300$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	9.60	4.93	2.47	1.31	0.69	0.41
2 processes per node	5.09	2.64	1.54	0.76	0.43	0.32
4 processes per node	2.58	1.40	0.73	0.45	0.33	0.36

(c) $N = 2{,}500$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	35.00	17.69	8.95	4.69	2.41	1.30
2 processes per node	20.47	9.62	4.87	2.54	1.36	0.97
4 processes per node	9.05	4.84	2.54	1.67	1.04	0.73

(d) $N = 4{,}100$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	94.40	47.72	24.13	12.42	6.33	3.32
2 processes per node	50.45	27.96	12.94	6.65	3.67	2.23
4 processes per node	25.45	12.53	6.45	3.95	2.43	1.68

(e) $N = 9{,}150$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	489.32	249.63	125.59	64.56	32.93	16.99
2 processes per node	291.32	142.66	75.03	39.34	19.49	10.08
4 processes per node	130.38	69.78	33.64	19.98	12.15	7.27

(f) $N = 17{,}000$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	1754.29	854.63	427.95	223.98	113.79	57.21
2 processes per node	939.21	503.39	255.32	128.41	63.56	36.32
4 processes per node	448.83	224.60	118.93	60.18	37.27	24.01

(g) $N = 33{,}900$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	N/A	N/A	2150.87	981.88	501.16	265.30
2 processes per node	N/A	N/A	1260.82	650.91	335.64	175.80
4 processes per node	N/A	N/A	559.66	290.41	162.27	101.13

(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	2528.92	1116.41
2 processes per node	N/A	N/A	N/A	N/A	1397.71	861.20
4 processes per node	N/A	N/A	N/A	N/A	719.32	364.10

(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	N/A
2 processes per node	N/A	N/A	N/A	N/A	N/A	4528.19
4 processes per node	N/A	N/A	N/A	N/A	N/A	2641.05

2 The Method

Affinity propagation functions by identifying similar datapoints in an iterative process [2]. The dataset is given as N datapoints of x and y coordinates, and the goal of the algorithm is to cluster groups of datapoints 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 datapoints i and j. We follow the default suggested in [2] by using the negative square of the Euclidean distance between the datapoints.

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 with significant memory usage. All matrices are split consistently across the p parallel processes by groups of adjacent columns. Our implementation uses the symmetry of S to compute as many quantities as possible by using only information that is local to each parallel process. This minimizes the number of parallel communications. As a result, we have only two `MPI_Allreduce` calls in each iteration. We use the programming language C and the OpenMPI implementation of MPI.

Since affinity propagation is based on matrix calculations, it has relatively large memory requirements. This can be seen concretely in Table 2 (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 dataset with $N = 126{,}700$ datapoints 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. Table 2 (b) shows the memory usage observed for our code. We observe that the memory required in reality is more predicted, but within reason for some smaller variables and required libraries; it is clear that we did not overlook any large arrays in our prediction. The memory usage is also stable over time, thus confirming that the code does not have a memory leak.

For testing purposes, we use a synthetic dataset that we can create in any desired size N. Moreover, this dataset is designed to let us control the true clusters. This allows us to check that the algorithm converged to the correct solution, independent of the number of parallel processes. The design of the synthetic dataset and its properties are discussed in detail in [1]. We run the affinity propagation algorithm with default numerical parameters suggested by [2] of `maxits` = 1,000, `convits` = 100, and damping parameter $\lambda = 0.9$.

Table 2: Memory usage on hpc using OpenMPI 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 OpenMPI

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	169	152	151	151	N/A	N/A	N/A	N/A
2,500	274	204	177	164	160	N/A	N/A	N/A
4,100	516	325	237	195	175	178	327	385
9,150	2,050	1,092	621	387	272	227	352	397
17,000	6,752	3,444	1,797	974	566	374	425	434
33,900	N/A	N/A	6,722	3,437	1,797	990	733	589
65,250	N/A	N/A	N/A	N/A	6,248	3,216	1,846	1,146
126,700	N/A	N/A	N/A	N/A	N/A	N/A	6,071	3,260

3 Performance Studies on hpc

The serial run times for the larger datasets observed in Table 1 bring out one key motivation for parallel computing: 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. More precisely, 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 for a fixed problem size is known as strong scalability of parallel code.

Table 3 lists the results of a performance study for strong scalability. Each row lists the results for one problem size parametrized by the number of datapoints N. Each column corresponds to the number of parallel processes p used in the run. The runs for Table 3 distribute these processes as widely as possible over the available nodes. That is, each process is run on a different node up to the available number of 32 nodes. In other words, up to $p = 32$, three of the four cores available on each node are idling, and only one core performs calculations. For $p = 64$ and $p = 128$, this cannot be accommodated on 32 nodes, thus 2 processes run on each node for $p = 64$ and 4 processes per node for $p = 128$. Comparing adjacent columns in the raw timing data in Table 3 (a) indicates that using twice as many processes speeds up the code by nearly a factor of two at least for all larger datasets. To quantify this more clearly, the speedup in Table 3 (b) is computed, which shows near-optimal with $S_p \approx p$ for all cases up to $p = 32$ which is expressed in terms of efficiency $0.84 \leq E_p \leq 1$ in Table 3 (c) for all but the two smallest datasets.

The customary visualizations of speedup and efficiency are presented in Figure 1 (a) and (b), respectively, for four intermediate values of N. Figure 1 (a) shows very clearly the very good speedup up to $p = 32$ parallel processes for all cases shown. The efficiency plotted in Figure 1 (b) is directly derived from the speedup, but the plot is still useful because it can better bring out any interesting features for small values of p that are hard to tell in a speedup plot. Here, we notice that the variability of the results for small p is visible. It is customary in results for 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 datasets seen in Table 3 (b) is expected. To see this clearly, it is vital to have the precise data in Table 3 (b) and (c) available and not just their graphical representation in Figure 1.

The conclusions discussed so far apply to up to $p = 32$ parallel processes. In each case, only 1 parallel process is run on each node with the other three cores available to handle all other operating system or other duties. For $p = 64$ and $p = 128$, 2 or 4 processes share each node necessarily, as only 32 nodes are available. Thus, one expects slightly degraded performance as we go from $p = 32$ to $p = 64$ and $p = 128$. This is borne out by all data in Table 3 as well as clearly visible in Figures 1 (a) and (b) for $p > 32$. However, the times in Table 3 (a) for all larger datasets clearly demonstrate an improvement by using more cores just not at the optimal rate of halving the wall clock time as p doubles.

To analyze the impact of using more than one core per node, we run 2 processes per node in Table 4 and Figure 2, and we run 4 processes per node in Table 5 and Figure 3, wherever possible. That is, for $p = 128$ in Table 4 and Figure 2, entries require 4 processes per node since only 32 nodes are available. On the other hand, in Table 5 and Figure 3, $p = 1$ is always computed on a dedicated node, i.e., running the entire job on a single process on a single node, and $p = 2$ is computed using a two-process job running on a single node. The results in the efficiency plots of Figures 2 (b) and 3 (b) show clearly that there is a significant loss of efficiency when going from $p = 1$ (always on a dedicated node) to $p = 2$ (with both processes on one node) to $p = 4$ (with 4 processes on one node).

The results presented so far indicate clearly the well-known conclusion that best performance improvements, in the sense of halving the time when doubling the number of processes, are achieved by only running one parallel process on each node. However, for production runs, we are not interested in this improvement being optimal, but we are interested in the run time being the smallest on a given number of nodes. Thus, given a fixed number of nodes, the question is if one should run 1, 2, or 4 processes per node. This is answered by the data organized in the form of Table 1 in the Introduction, and we saw clearly that it is well worthwhile to use all available cores for fastest absolute run times.

Table 3: Performance on hpc using OpenMPI by number of processes used with 1 process per node except for $p = 64$ which uses 2 processes per node and $p = 128$ which uses 4 processes per node. N/A indicates that the case required more memory than available.

(a) Wall clock time in seconds

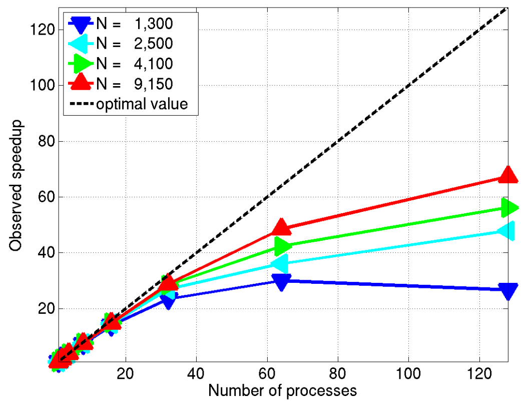
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.73	0.91	0.44	0.22	0.13	0.08	0.07	0.13
1,300	9.60	4.93	2.47	1.31	0.69	0.41	0.32	0.36
2,500	35.00	17.69	8.95	4.69	2.41	1.30	0.97	0.73
4,100	94.40	47.72	24.13	12.42	6.33	3.32	2.23	1.68
9,150	489.32	249.63	125.59	64.56	32.93	16.99	10.08	7.27
17,000	1754.29	854.63	427.95	223.98	113.79	57.21	36.32	24.01
33,900	N/A	N/A	2150.87	981.88	501.16	265.30	175.80	101.13
65,250	N/A	N/A	N/A	N/A	2528.92	1116.41	861.20	364.10
126,700	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

(b) Observed speedup S_p

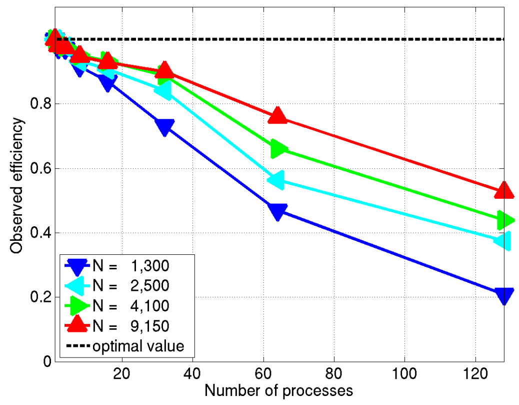
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.0000	1.9011	3.9318	7.8636	13.3077	21.6250	24.7143	13.3077
1,300	1.0000	1.9473	3.8866	7.3282	13.9130	23.4146	30.0000	26.6667
2,500	1.0000	1.9785	3.9106	7.4627	14.5228	26.9231	36.0825	47.9452
4,100	1.0000	1.9782	3.9121	7.6006	14.9131	28.4337	42.3318	56.1905
9,150	1.0000	1.9602	3.8962	7.5793	14.8594	28.8005	48.5437	67.3067
17,000	1.0000	2.0527	4.0993	7.8324	15.4169	30.6640	48.3009	73.0650
33,900	N/A	N/A	4.0000	8.7623	17.1671	32.4292	48.9390	85.0735
65,250	N/A	N/A	N/A	N/A	16.0000	36.2436	46.9841	111.1308
126,700	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

(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	0.9505	0.9830	0.9830	0.8317	0.6758	0.3862	0.1040
1,300	1.0000	0.9736	0.9717	0.9160	0.8696	0.7317	0.4688	0.2083
2,500	1.0000	0.9893	0.9777	0.9328	0.9077	0.8413	0.5638	0.3746
4,100	1.0000	0.9891	0.9780	0.9501	0.9321	0.8886	0.6614	0.4390
9,150	1.0000	0.9801	0.9740	0.9474	0.9287	0.9000	0.7585	0.5258
17,000	1.0000	1.0263	1.0248	0.9790	0.9636	0.9583	0.7547	0.5708
33,900	N/A	N/A	1.0000	1.0953	1.0729	1.0134	0.7647	0.6646
65,250	N/A	N/A	N/A	N/A	1.0000	1.1326	0.7341	0.8682
126,700	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 1: Performance on hpc using OpenMPI by number of processes used with 1 process per node except for $p = 64$ which uses 2 processes per node and $p = 128$ which uses 4 processes per node.

Table 4: Performance on hpc using OpenMPI by number of processes used with 2 processes per node except for $p = 1$ which uses 1 process per node and $p = 128$ which uses 4 processes per node. Also, data marked by an asterisk do not use 2 processes per node but are copied from the previous table to allow for a comparison here. N/A indicates that the case required more memory than available.

(a) Wall clock time in seconds

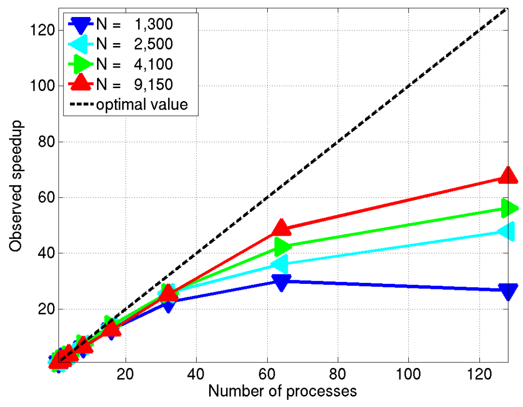
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.73	0.92	0.46	0.21	0.12	0.09	0.07	0.13
1,300	9.60	5.09	2.64	1.54	0.76	0.43	0.32	0.36
2,500	35.00	20.47	9.62	4.87	2.54	1.36	0.97	0.73
4,100	94.40	50.45	27.96	12.94	6.65	3.67	2.23	1.68
9,150	489.32	291.32	142.66	75.03	39.34	19.49	10.08	7.27
17,000	1754.29	939.21	503.39	255.32	128.41	63.56	36.32	24.01
33,900	N/A	N/A	*2150.87	1260.82	650.91	335.64	175.80	101.13
65,250	N/A	N/A	N/A	N/A	*2528.92	1397.71	861.20	364.10
126,700	N/A	N/A	N/A	N/A	N/A	N/A	4528.19	2641.05

(b) Observed speedup S_p

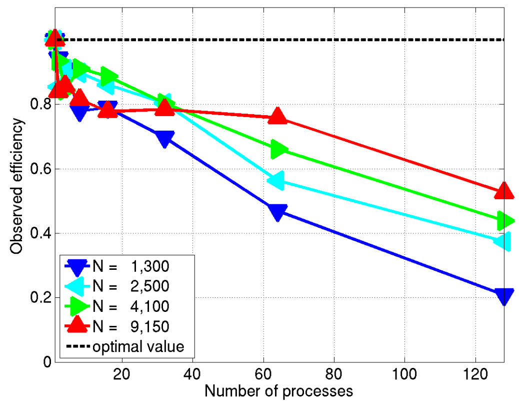
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.0000	1.8804	3.7609	8.2381	14.4167	19.2222	24.7143	13.3077
1,300	1.0000	1.8861	3.6364	6.2338	12.6316	22.3256	30.0000	26.6667
2,500	1.0000	1.7098	3.6383	7.1869	13.7795	25.7353	36.0825	47.9452
4,100	1.0000	1.8712	3.3763	7.2952	14.1955	25.7221	42.3318	56.1905
9,150	1.0000	1.6797	3.4300	6.5217	12.4382	25.1062	48.5437	67.3067
17,000	1.0000	1.8678	3.4850	6.8709	13.6616	27.6005	48.3009	73.0650
33,900	N/A	N/A	*4.0000	6.8237	13.2176	25.6331	48.9390	85.0735
65,250	N/A	N/A	N/A	N/A	*16.0000	28.9493	46.9841	111.1308
126,700	N/A	N/A	N/A	N/A	N/A	N/A	64.0000	109.7307

(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	0.9402	0.9402	1.0298	0.9010	0.6007	0.3862	0.1040
1,300	1.0000	0.9430	0.9091	0.7792	0.7895	0.6977	0.4688	0.2083
2,500	1.0000	0.8549	0.9096	0.8984	0.8612	0.8042	0.5638	0.3746
4,100	1.0000	0.9356	0.8441	0.9119	0.8872	0.8038	0.6614	0.4390
9,150	1.0000	0.8398	0.8575	0.8152	0.7774	0.7846	0.7585	0.5258
17,000	1.0000	0.9339	0.8712	0.8589	0.8539	0.8625	0.7547	0.5708
33,900	N/A	N/A	*1.0000	0.8530	0.8261	0.8010	0.7647	0.6646
65,250	N/A	N/A	N/A	N/A	*1.0000	0.9047	0.7341	0.8682
126,700	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	0.8573



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 2: Performance on hpc using OpenMPI by number of processes used with 2 processes per node except for $p = 1$ which uses 1 process per node and $p = 128$ which uses 4 processes per node.

Table 5: Performance on hpc using OpenMPI 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. Also, data marked by an asterisk do not use 4 processes per node but are copied from the previous tables to allow for a comparison here. N/A indicates that the case required more memory than available.

(a) Wall clock time in seconds

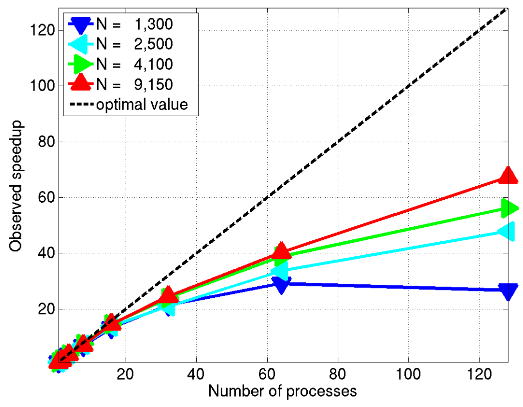
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.73	0.92	0.45	0.22	0.13	0.10	0.11	0.13
1,300	9.60	5.09	2.58	1.40	0.73	0.45	0.33	0.36
2,500	35.00	20.47	9.05	4.84	2.54	1.67	1.04	0.73
4,100	94.40	50.45	25.45	12.53	6.45	3.95	2.43	1.68
9,150	489.32	291.32	130.38	69.78	33.64	19.98	12.15	7.27
17,000	1754.29	939.21	448.83	224.60	118.93	60.18	37.27	24.01
33,900	N/A	N/A	*2150.87	*1260.82	559.66	290.41	162.27	101.13
65,250	N/A	N/A	N/A	N/A	*2528.92	*1397.71	719.32	364.10
126,700	N/A	N/A	N/A	N/A	N/A	N/A	*4528.19	2641.05

(b) Observed speedup S_p

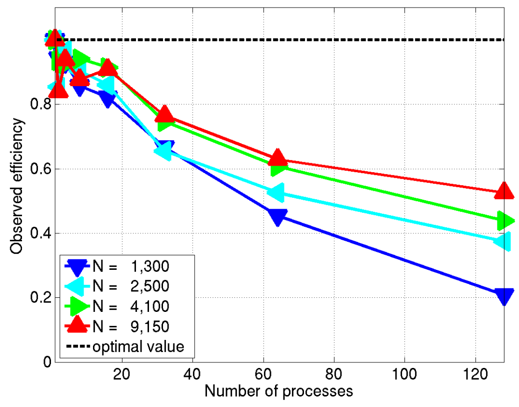
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.0000	1.8804	3.8444	7.8636	13.3077	17.3000	15.7273	13.3077
1,300	1.0000	1.8861	3.7209	6.8571	13.1507	21.3333	29.0909	26.6667
2,500	1.0000	1.7098	3.8674	7.2314	13.7795	20.9581	33.6538	47.9452
4,100	1.0000	1.8712	3.7092	7.5339	14.6357	23.8987	38.8477	56.1905
9,150	1.0000	1.6797	3.7530	7.0123	14.5458	24.4905	40.2733	67.3067
17,000	1.0000	1.8678	3.9086	7.8107	14.7506	29.1507	47.0698	73.0650
33,900	N/A	N/A	*4.0000	6.8237	15.3727	29.6253	53.0195	85.0735
65,250	N/A	N/A	N/A	N/A	*16.0000	28.9493	56.2513	111.1308
126,700	N/A	N/A	N/A	N/A	N/A	N/A	64.0000	109.7307

(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	0.9402	0.9611	0.9830	0.8317	0.5406	0.2457	0.1040
1,300	1.0000	0.9430	0.9302	0.8571	0.8219	0.6667	0.4545	0.2083
2,500	1.0000	0.8549	0.9669	0.9039	0.8612	0.6549	0.5258	0.3746
4,100	1.0000	0.9356	0.9273	0.9417	0.9147	0.7468	0.6070	0.4390
9,150	1.0000	0.8398	0.9383	0.8765	0.9091	0.7653	0.6293	0.5258
17,000	1.0000	0.9339	0.9771	0.9763	0.9219	0.9110	0.7355	0.5708
33,900	N/A	N/A	*1.0000	0.8530	0.9608	0.9258	0.8284	0.6646
65,250	N/A	N/A	N/A	N/A	*1.0000	0.9047	0.8789	0.8682
126,700	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	0.8573



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 3: Performance on hpc using OpenMPI 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.

A Performance Studies on kali

This appendix summarizes results of analogous studies to the previous sections performed on the cluster kali purchased in 2003. This cluster originally had 32 nodes, each with two (single-core) processors (Intel Xeon 2.0 GHz with 512 kB cache) and 1 GB of memory (except the storage node with 4 GB of memory), connected by a Myrinet interconnect network. Only 27 of the 32 nodes are connected by the Myrinet network at present (2008). Hence, only 16 nodes are available for parallel performance study when considering only powers of 2 for convenience.

Table 6 summarizes the predicted and observed memory usage of the code. We see that $N = 17{,}000$ is the largest dataset that can be accommodated on 16 nodes with 1 GB of memory each.

Table 7 is a summary table of raw timing results analogous to Table 1 in the Introduction. We observe that the execution times in Table 1 are more than twice as fast as those recorded in Table 7 for corresponding entries, that is, for the entries with the same number of nodes and parallel processes per node. For instance, we see that kali can solve the case with $N = 17{,}000$ datapoints when using 16 nodes, and the best observed time is 191.77 seconds with 2 parallel processes on each node. By comparison, hpc requires about 63.56 seconds for this problem using 16 nodes and 2 processes per node. However, the optimal time on hpc, when using 16 nodes, is in fact 37.27 seconds when using all 4 cores on each node. This shows the benefit of having 4 cores per node concretely. Looking at the comparison between the machines in a different way, we see that only 4 nodes on hpc instead of 16 on kali are required to solve this problem in an even faster wall clock time of 118.93 seconds.

Table 8 and Figure 4 summarize and visualize the underlying performance results for the case of running only 1 process on each node except $p = 32$ with 2 processes and are analogous to Table 3 and Figure 1. We note here that the necessary memory for the largest dataset requires at least 8 nodes. Therefore, speedup is redefined to use only the available data as $S_p := 8T_8(N)/T_p(N)$ for the dataset with $N = 17{,}000$ datapoints. Comparing the corresponding efficiency data in Table 3 (c) with Table 8 (c), we notice that the efficiency demonstrated by kali is better than that seen in the new IBM machine up to $p = 16$ for all datasets where data is available. However, while considering this speedup result, we must recall that the new IBM machine completes the task in half the time as kali for every value of p.

Table 9 and Figure 5 summarize and visualize the performance results for the case of running 2 processes on each node except $p = 1$ with 1 process. Once more, we redefine speedup for the largest dataset to use only the available data as $S_p := 16T_{16}(N)/T_p(N)$. The efficiency plot in Figure 5 (b) demonstrates that the performance degradation occurs from $p = 1$ to $p = 2$. That is, it is associated with using both processors per node instead of one process only.

References

[1] Robin Blasberg and Matthias K. Gobbert. Clustering large datasets with parallel affinity propagation. Submitted.

[2] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, vol. 315, pp. 972–976, 2007.

[3] Matthias K. Gobbert. Parallel performance studies for an elliptic test problem. Technical Report HPCF–2008–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.

Table 6: Memory usage on kali in MB per process. Data is from runs with one process per node where available. 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. Cases marked by an asterisk were run on the storage node with 4 GB of memory.

(a) Predicted memory usage in MB per process						
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	6	3	1	1	< 1	< 1
1,300	39	19	10	5	2	1
2,500	143	72	36	18	9	4
4,100	385	192	96	48	24	12
9,150	1,916	958	479	240	120	60
17,000	6,615	3,307	1,654	827	413	207
33,900	26,303	13,152	6,576	3,288	1,644	822

(b) Observed memory usage on kali						
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	8	25	N/A	N/A	22	N/A
1,300	42	42	32	27	22	N/A
2,500	145	96	59	41	31	29
4,100	388	214	118	72	47	37
9,150	*1,921	*964	502	261	141	86
17,000	N/A	N/A	N/A	850	436	231

Table 7: Performance on kali. Wall clock time in seconds for the solution of problems with N data points using 1, 2, 4, 8, 16 compute nodes with 1 and 2 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
1 process per node	4.10	2.05	1.03	0.54	0.27
2 processes per node	2.19	1.14	0.60	0.28	0.18
(b) $N = 1{,}300$	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	23.15	11.45	5.80	2.92	1.50
2 processes per node	12.38	6.29	3.24	1.68	0.94
(c) $N = 2{,}500$	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	85.58	41.17	20.83	10.67	5.43
2 processes per node	44.01	22.14	11.50	5.94	3.17
(d) $N = 4{,}100$	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	221.17	109.58	55.77	28.64	14.40
2 processes per node	117.05	59.54	30.25	15.99	8.26
(e) $N = 9{,}150$	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	1157.88	N/A	288.61	147.42	75.13
2 processes per node	683.78	N/A	158.91	91.16	44.96
(f) $N = 17{,}000$	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	N/A	N/A	N/A	580.66	281.90
2 processes per node	N/A	N/A	N/A	421.01	191.77

Table 8: Performance on kali by number of processes used with 1 process per node except for $p = 32$ which uses 2 processes per node. N/A indicates that the case required more memory than available.

(a) Wall clock time in seconds

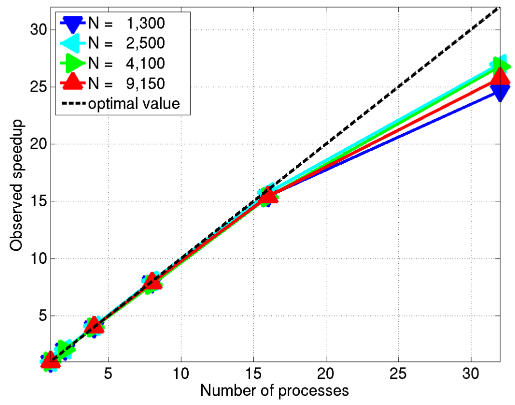
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	4.10	2.05	1.03	0.54	0.27	0.18
1,300	23.15	11.45	5.80	2.92	1.50	0.94
2,500	85.58	41.17	20.83	10.67	5.43	3.17
4,100	221.17	109.58	55.77	28.64	14.40	8.26
9,150	1157.88	N/A	288.61	147.42	75.13	44.96
17,000	N/A	N/A	N/A	580.66	281.90	191.77

(b) Observed speedup S_p

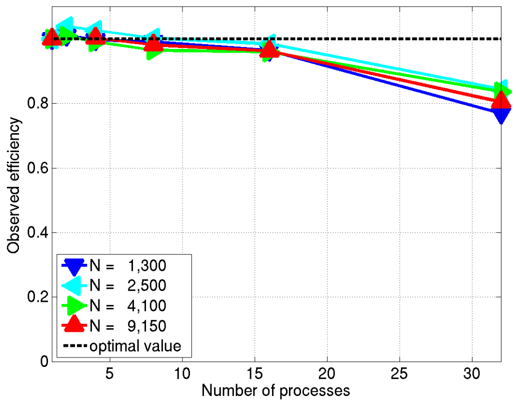
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	1.0000	2.0000	3.9806	7.5926	15.1852	22.7778
1,300	1.0000	2.0218	3.9914	7.9281	15.4333	24.6277
2,500	1.0000	2.0787	4.1085	8.0206	15.7606	26.9968
4,100	1.0000	2.0183	3.9658	7.7224	15.3590	26.7760
9,150	1.0000	N/A	4.0119	7.8543	15.4117	25.7536
17,000	N/A	N/A	N/A	8.0000	16.4785	24.2232

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	1.0000	1.0000	0.9951	0.9491	0.9491	0.7118
1,300	1.0000	1.0109	0.9978	0.9910	0.9646	0.7696
2,500	1.0000	1.0393	1.0271	1.0026	0.9850	0.8437
4,100	1.0000	1.0092	0.9914	0.9653	0.9599	0.8368
9,150	1.0000	N/A	1.0030	0.9818	0.9632	0.8048
17,000	N/A	N/A	N/A	1.0000	1.0299	0.7570



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 4: Performance on kali by number of processes used with 1 process per node except for $p = 32$ which uses 2 processes per node.

Table 9: Performance on kali by number of processes used with 2 processes per node except for $p = 1$ which uses 1 process per node.

(a) Wall clock time in seconds

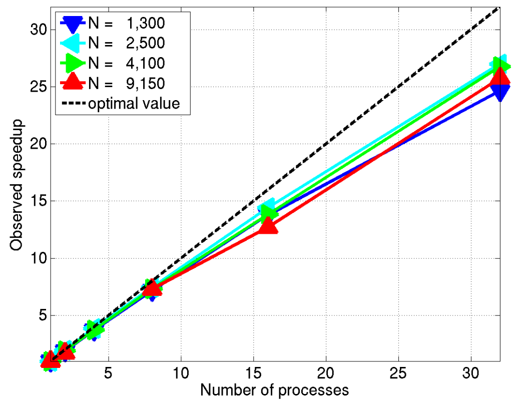
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	4.10	2.19	1.14	0.60	0.28	0.18
1,300	23.15	12.38	6.29	3.24	1.68	0.94
2,500	85.58	44.01	22.14	11.50	5.94	3.17
4,100	221.17	117.05	59.54	30.25	15.99	8.26
9,150	1157.88	683.78	N/A	158.91	91.16	44.96
17,000	N/A	N/A	N/A	N/A	421.01	191.77

(b) Observed speedup S_p

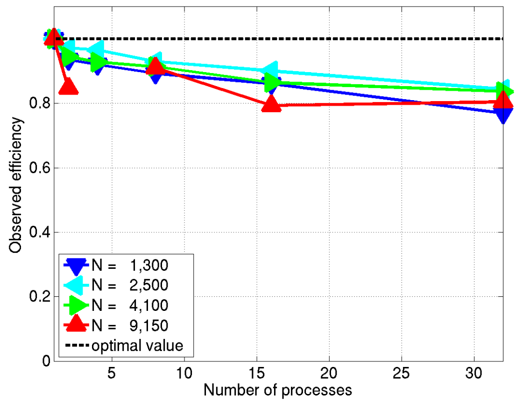
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	1.0000	1.8721	3.5965	6.8333	14.6429	22.7778
1,300	1.0000	1.8700	3.6804	7.1451	13.7798	24.6277
2,500	1.0000	1.9446	3.8654	7.4417	14.4074	26.9968
4,100	1.0000	1.8895	3.7146	7.3114	13.8318	26.7760
9,150	1.0000	1.6934	N/A	7.2864	12.7016	25.7536
17,000	N/A	N/A	N/A	N/A	16.0000	35.1262

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
500	1.0000	0.9361	0.8991	0.8542	0.9152	0.7118
1,300	1.0000	0.9350	0.9201	0.8931	0.8612	0.7696
2,500	1.0000	0.9723	0.9664	0.9302	0.9005	0.8437
4,100	1.0000	0.9448	0.9287	0.9139	0.8645	0.8368
9,150	1.0000	0.8467	N/A	0.9108	0.7939	0.8048
17,000	N/A	N/A	N/A	N/A	1.0000	1.0977



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 5: Performance on kali by number of processes used with 2 processes per node except for $p = 1$ which uses 1 process per node.