MVAPICH2 vs. OpenMPI 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 data points in an iterative process. Its structure allows for taking full advantage of parallel computing to enable the solution of larger problems and to solve them faster than possible in serial. We present a memory-optimal implementation with minimal number of communication commands and demonstrate its excellent scalability. Additionally, we present a comparison of two implementations of MPI that demonstrate that MVAPICH2 exhibits better scalability up to larger numbers of parallel processes than OpenMPI for this code.

1 Introduction

Affinity propagation is a relatively new clustering algorithm introduced by Frey and Dueck [3] that functions by identifying similar data points 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 data points in the given data set, 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 data set with $N = 17{,}000$ data points 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 [4].

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 in detail the parallel scalability results using the MVAPICH2 implementation of MPI and provides the underlying data for the following summary results. Table 1 summarizes the key results by giving the wall clock time (total time to execute the code) in seconds. We consider nine progressively larger data sets, as indicated by the number of data points 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. One realizes the advantage of parallel computing for a case with $N = 17{,}000$ data points requiring 6.6 GB of memory: The serial run time in the upper-left entry of the sub-table of about 30 minutes (1798.04 seconds) is reduced to about 15 seconds in the lower-right entry of the sub-table using 128 parallel processes. Yet, the true advantage of parallelizing the algorithm is evident for a problem with $N = 126{,}700$ data points 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 45 minutes (2720.27 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 the largest number of nodes (i.e., $p = 32$) for the smallest data sets. 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.

(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 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 are still excellent and 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 shows that the architecture of the nodes 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) as well as the purchase of dual-core processors (as opposed to single-core processors).

Finally, Section 4 collects and extends analogous results from [2] obtained using the OpenMPI implementation of MPI. It turns out that for this code, MVAPICH2 exhibits better scalability than OpenMPI up to large numbers of parallel processes.

Table 1: 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

2 The Method

Affinity propagation functions by identifying similar data points in an iterative process [3]. The data set is given as N data points of x and y coordinates, 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 [3] by using the negative square of the Euclidean distance between the 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 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 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 both the MVAPICH2 and OpenMPI implementations of MPI, extending [2] that used only OpenMPI.

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. As can be seen, 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. Table 2 (b) shows the memory usage observed for our code using MVAPICH2. The observed memory usage is only sightly higher than the predicted values. This confirms that we did not overlook any large arrays in our prediction.

For testing purposes, we use a synthetic data set that we can create in any desired size N. Moreover, this data set 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 data set and its properties are discussed in detail in [1]. We run the affinity propagation algorithm with default numerical parameters suggested by [3] of `maxits` = 1,000, `convits` = 100, and damping parameter $\lambda = 0.9$.

Table 2: 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

3 Performance Studies using MVAPICH2

The serial run times for the larger data sets 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. The optimal value of this speedup 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 the optimal value of $E_p = 1$. The 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 data points 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 but some of the $p = 64$ and $p = 128$ cases. To quantify this more clearly, the speedup in Table 3 (b) is computed, which shows near-optimal with $S_p \approx p$ for all but roughly one case up to $p = 32$. This is expressed in terms of efficiency $0.81 \leq E_p \leq 1$ in Table 3 (c) for all p values $9{,}150 \leq N \leq 65{,}250$.

The customary visualizations of speedup and efficiency are presented in Figure 1 (a) and (b), respectively, for five 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, but excellent efficiency throughout. 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 data sets seen in Table 3 (b) for $N \leq 33{,}900$ 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.

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. More specifically, $p = 1$ is always computed on a dedicated node, i.e., running the entire job on a single process on a single node. 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, $p = 2$ 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 that there is some 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), when compared to the the results in Figure 1 (b). However, it is remarkable how little loss of efficiency there is for this combination of code, MPI implementation, and hardware.

The analysis of the data presented so far shows that the best performance improvement, in the sense of halving the time when doubling the number of processes, is achieved by only running one parallel process on each node, even though the advantage was very small by comparison. 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 MVAPICH2 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

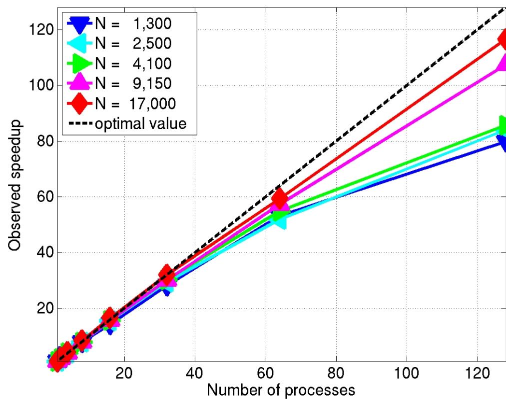
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.71	0.87	0.45	0.22	0.12	0.08	0.06	0.05
1,300	9.57	4.77	2.45	1.23	0.67	0.34	0.18	0.12
2,500	35.29	17.58	8.96	4.48	2.28	1.20	0.68	0.42
4,100	95.99	47.83	23.98	12.14	6.13	3.18	1.75	1.12
9,150	498.37	251.72	126.76	63.16	31.81	16.62	8.70	4.64
17,000	1798.04	864.00	424.48	214.80	109.19	56.27	30.32	15.43
33,900	N/A	N/A	2287.98	1004.23	492.52	255.92	153.58	78.81
65,250	N/A	N/A	N/A	N/A	2593.53	1291.01	742.32	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

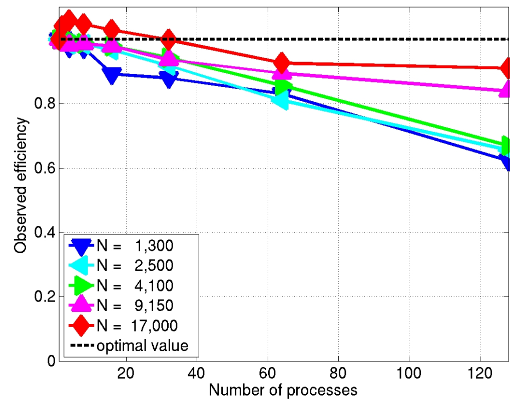
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.0000	1.9655	3.8000	7.7727	14.2500	21.3750	28.5000	34.2000
1,300	1.0000	2.0063	3.9061	7.7805	14.2836	28.1471	53.1667	79.7500
2,500	1.0000	2.0074	3.9386	7.8772	15.4781	29.4083	51.8971	84.0238
4,100	1.0000	2.0069	4.0029	7.9069	15.6591	30.1855	54.8514	85.7054
9,150	1.0000	1.9799	3.9316	7.8906	15.6671	29.9862	57.2839	107.4073
17,000	1.0000	2.0811	4.2359	8.3708	16.4671	31.9538	59.3021	116.5288
33,900	N/A	N/A	4.0000	9.1134	18.5818	35.7609	59.5906	116.1264
65,250	N/A	N/A	N/A	N/A	16.0000	32.1426	55.9011	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	0.9828	0.9500	0.9716	0.8906	0.6680	0.4453	0.2672
1,300	1.0000	1.0031	0.9765	0.9726	0.8927	0.8796	0.8307	0.6230
2,500	1.0000	1.0037	0.9847	0.9847	0.9674	0.9190	0.8109	0.6564
4,100	1.0000	1.0034	1.0007	0.9884	0.9787	0.9433	0.8571	0.6696
9,150	1.0000	0.9899	0.9829	0.9863	0.9792	0.9371	0.8951	0.8391
17,000	1.0000	1.0405	1.0590	1.0463	1.0292	0.9986	0.9266	0.9104
33,900	N/A	N/A	1.0000	1.1392	1.1614	1.1175	0.9311	0.9072
65,250	N/A	N/A	N/A	N/A	1.0000	1.0045	0.8735	0.8053
126,700	N/A	N/A	N/A	N/A	N/A	1.0000	0.6855	0.5662



(a) Observed speedup S_p

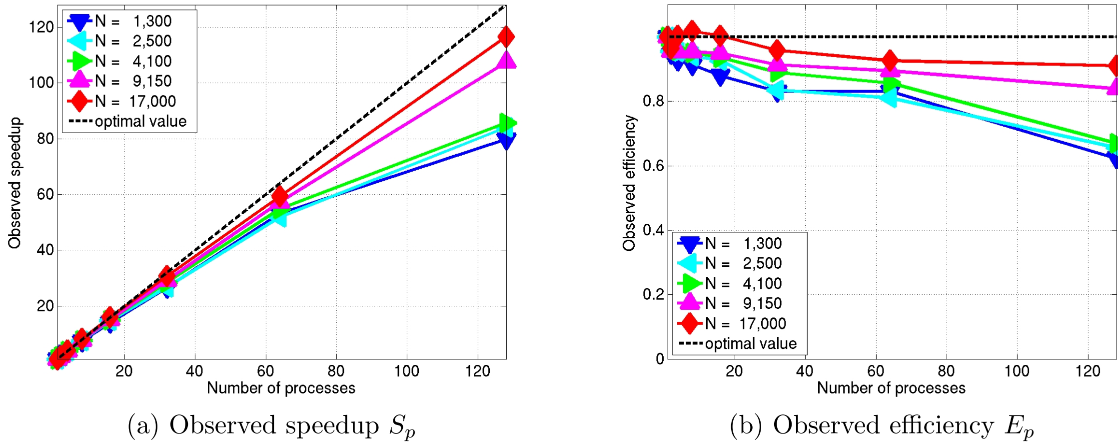


(b) Observed efficiency E_p

Figure 1: Performance on `hpc` using MVAPICH2 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 MVAPICH2 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	1.71	0.91	0.47	0.22	0.13	0.08	0.06	0.05
1,300	9.57	5.05	2.58	1.31	0.68	0.36	0.18	0.12
2,500	35.29	18.48	9.25	4.69	2.38	1.32	0.68	0.42
4,100	95.99	49.72	25.06	12.62	6.41	3.37	1.75	1.12
9,150	498.37	260.85	130.74	65.31	32.83	17.05	8.70	4.64
17,000	1798.04	932.81	448.18	220.77	111.99	58.64	30.32	15.43
33,900	N/A	N/A	*2287.98	1223.54	553.48	287.41	153.58	78.81
65,250	N/A	N/A	N/A	N/A	*2593.53	1595.91	742.32	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	3.6383	7.7727	13.1538	21.3750	28.5000	34.2000
1,300	1.0000	1.8950	3.7093	7.3053	14.0735	26.5833	53.1667	79.7500
2,500	1.0000	1.9096	3.8151	7.5245	14.8277	26.7348	51.8971	84.0238
4,100	1.0000	1.9306	3.8304	7.6062	14.9750	28.4837	54.8514	85.7054
9,150	1.0000	1.9106	3.8119	7.6308	15.1803	29.2299	57.2839	107.4073
17,000	1.0000	1.9276	4.0119	8.1444	16.0554	30.6623	59.3021	116.5288
33,900	N/A	N/A	*4.0000	7.4799	16.5352	31.8427	59.5906	116.1264
65,250	N/A	N/A	N/A	N/A	*16.0000	26.0018	55.9011	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	0.9096	0.9716	0.8221	0.6680	0.4453	0.2672
1,300	1.0000	0.9475	0.9273	0.9132	0.8796	0.8307	0.8307	0.6230
2,500	1.0000	0.9548	0.9538	0.9406	0.9267	0.8355	0.8109	0.6564
4,100	1.0000	0.9653	0.9576	0.9508	0.9359	0.8901	0.8571	0.6696
9,150	1.0000	0.9553	0.9530	0.9539	0.9488	0.9134	0.8951	0.8391
17,000	1.0000	0.9638	1.0030	1.0181	1.0035	0.9582	0.9266	0.9104
33,000	N/A	N/A	*1.0000	0.9350	1.0335	0.9951	0.9311	0.9072
65,250	N/A	N/A	N/A	N/A	*1.0000	0.8126	0.8735	0.8053
126,700	N/A	N/A	N/A	N/A	N/A	*1.0000	0.6855	0.5662



(a) Observed speedup S_p

(b) Observed efficiency E_p

Figure 2: Performance on hpc using MVAPICH2 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 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. 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

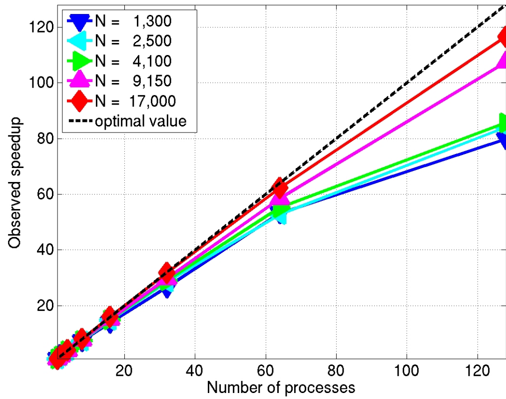
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
500	1.71	0.91	0.53	0.22	0.14	0.08	0.06	0.05
1,300	9.57	5.05	2.64	1.32	0.69	0.36	0.18	0.12
2,500	35.29	18.48	9.20	4.77	2.39	1.24	0.67	0.42
4,100	95.99	49.72	25.11	12.78	6.43	3.31	1.74	1.12
9,150	498.37	260.85	130.42	64.99	32.86	16.78	8.53	4.64
17,000	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

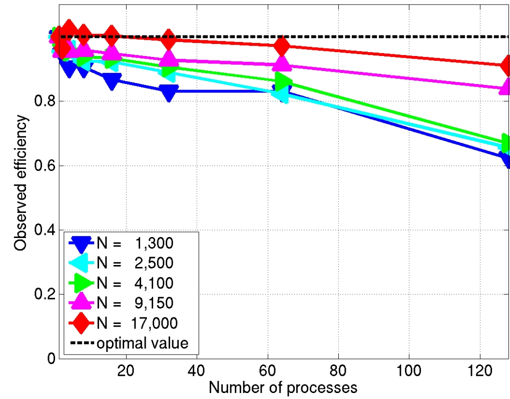
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
500	1.0000	1.8791	3.2264	7.7727	12.2143	21.3750	28.5000	34.2000
1,300	1.0000	1.8950	3.6250	7.2500	13.8696	26.5833	53.1667	79.7500
2,500	1.0000	1.9096	3.8359	7.3983	14.7657	28.4597	52.6716	84.0238
4,100	1.0000	1.9306	3.8228	7.5110	14.9285	29.0000	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



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 3: 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.

4 Performance Studies using OpenMPI

This section compares the results obtained by using the OpenMPI implementation of MPI to the results in the earlier sections obtained with MVAPICH2. The tables in this section repeat the results from [2] for convenience, while the figures are extended by results for $N = 17{,}000$. Tables 6–10 and Figures 4–6 are the OpenMPI equivalents of Tables 1–5 and Figures 1–3, respectively.

Comparing Table 1 for MVAPICH2 with Table 6 for OpenMPI shows that the wall clock times for the vast majority of runs with 8 or more nodes are faster using MVAPICH2. For smaller numbers of compute nodes as well as for the largest two data sets, the results are somewhat mixed, but, for the most part, the times are not significantly different. The observed memory usages shown in Table 2 and Table 7 demonstrate that MVAPICH2 uses less memory, and, in some cases, significantly less.

A comparison of the MVAPICH2 Figures 1–3 with their equivalent OpenMPI Figures 4–6 illustrates that the observed speedup results for MVAPICH2 are far closer to the optimal linear speedup than the OpenMPI

Table 6: 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

results. In fact, the speedups for OpenMPI in Figures 4–6 are good up to $p = 32$ parallel processes for the cases shown, but there is a significant loss of efficiency for $p = 64$ and 128. By contrast, the MPVAPICH2 speedups in Figures 1–3 are good up to $p = 64$ parallel processes for all data sets and up to 128 for the two largest data set sizes presented in the figures.

References

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

[2] Robin Blasberg and Matthias K. Gobbert. Parallel performance studies for a clustering algorithm. Technical Report HPCF–2008–5, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.

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

[4] 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 7: 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

Table 8: 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

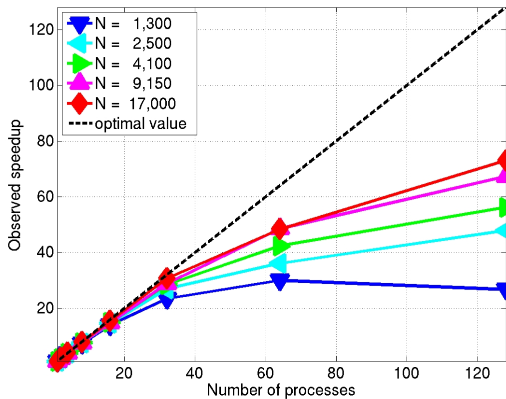
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
500	1.73	0.91	0.44	0.22	0.13	0.08	0.07	0.13
1,300	9.60	4.93	2.47	1.31	0.69	0.41	0.32	0.36
2,500	35.00	17.69	8.95	4.69	2.41	1.30	0.97	0.73
4,100	94.40	47.72	24.13	12.42	6.33	3.32	2.23	1.68
9,150	489.32	249.63	125.59	64.56	32.93	16.99	10.08	7.27
17,000	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

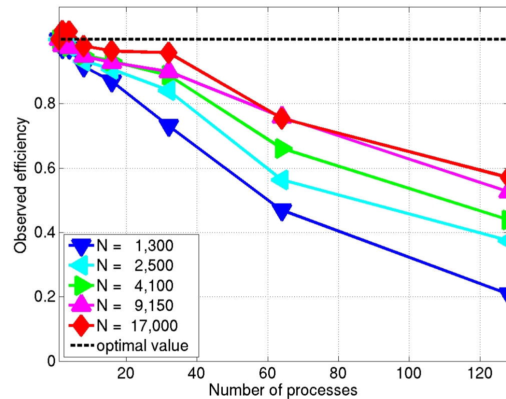
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
500	1.0000	1.9011	3.9318	7.8636	13.3077	21.6250	24.7143	13.3077
1,300	1.0000	1.9473	3.8866	7.3282	13.9130	23.4146	30.0000	26.6667
2,500	1.0000	1.9785	3.9106	7.4627	14.5228	26.9231	36.0825	47.9452
4,100	1.0000	1.9782	3.9121	7.6006	14.9131	28.4337	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 4: 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 9: 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

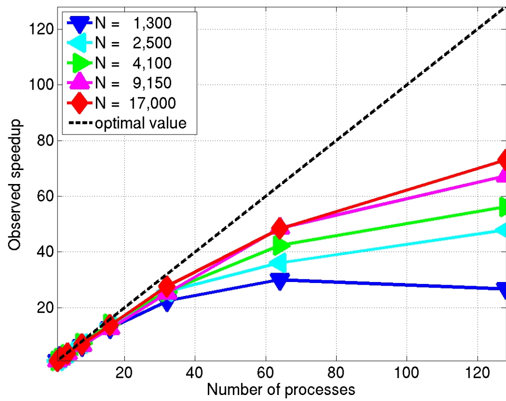
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.73	0.92	0.46	0.21	0.12	0.09	0.07	0.13
1,300	9.60	5.09	2.64	1.54	0.76	0.43	0.32	0.36
2,500	35.00	20.47	9.62	4.87	2.54	1.36	0.97	0.73
4,100	94.40	50.45	27.96	12.94	6.65	3.67	2.23	1.68
9,150	489.32	291.32	142.66	75.03	39.34	19.49	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

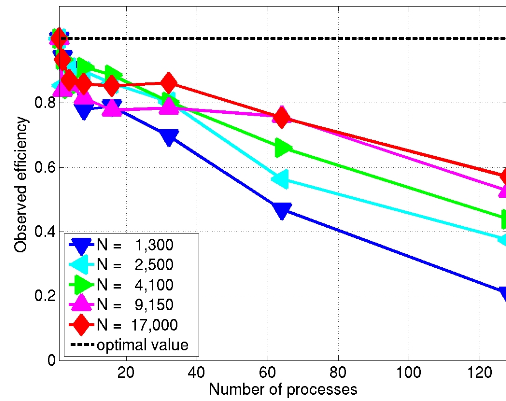
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
500	1.0000	1.8804	3.7609	8.2381	14.4167	19.2222	24.7143	13.3077
1,300	1.0000	1.8861	3.6364	6.2338	12.6316	22.3256	30.0000	26.6667
2,500	1.0000	1.7098	3.6383	7.1869	13.7795	25.7353	36.0825	47.9452
4,100	1.0000	1.8712	3.3763	7.2952	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

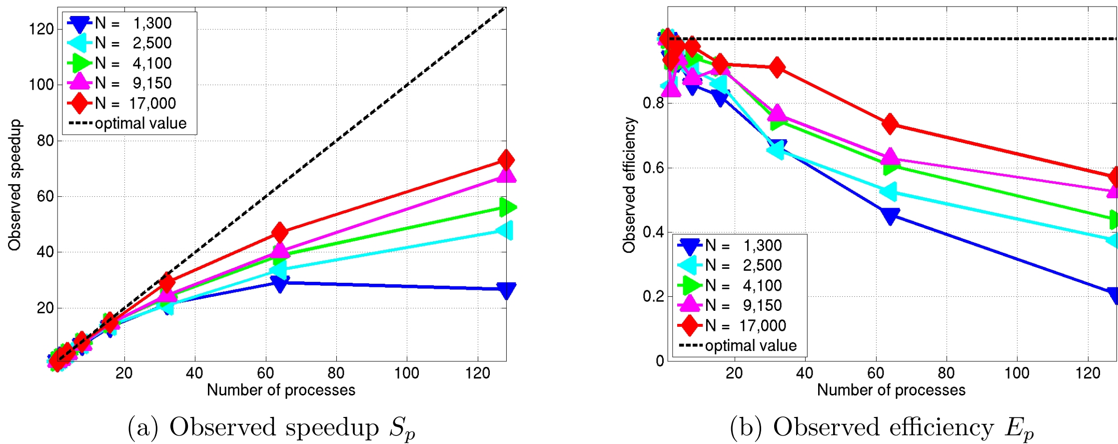
Figure 5: Performance on `hpc` using OpenMPI by number of processes used with 2 processes per node except for $p = 1$ which uses 1 process per node and $p = 128$ which uses 4 processes per node.

Table 10: 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 6: 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.