Parallel Algorithms for Bayesian Indoor Positioning Systems

Konstantinos Kleisouris, Richard P. Martin
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854
{kkonst,rmartin}@cs.rutgers.edu

Abstract

We present two parallel algorithms and their Unified Parallel C implementations for Bayesian indoor positioning systems. Our approaches are founded on Markov Chain Monte Carlo simulations. We evaluated two basic partitioning schemes: inter-chain partitioning which distributes entire Markov chains to different processors, and intra-chain which distributes a single chain across processors. Evaluations on a 16-node symmetric multiprocessor, a 4-node cluster comprising of quad processors, and a 16 single-processor-node cluster, suggest that for short chains intra-chain scales well on the first two platforms with speedups of up to 12. On the other hand, inter-chain gives speedups of 12 only for very long chains, sometimes of up to 60,000 iterations, on all three platforms. We used the LogGP model to analyze our algorithms and predict their performance. Model predictions for inter-chain are within 5% of the actual execution time, while for intra-chain they are 7%-25% less due to load imbalance not captured in the model.

1 Introduction

Indoor location estimation is an important problem for wireless networks, such as IEEE 802.11 (Wi-Fi), as it allows the tracking of devices like laptop computers, handheld devices, and electronic badges inside places such as stores, hospitals and factories. Recently solutions for indoor positioning have been proposed that rely on Bayesian networks [7, 8, 15]. These networks incorporate several modalities like received signal strength (RSS) and angle of arrival (AoA) of the signal to provide location estimates.

In a previous study [11] we implemented several Bayesian inference methods for these networks using Markov Chain Monte Carlo (MCMC) walks [10]. MCMC is essentially a sampling method that approximates the true probability distribution of the unknown variables in the network. In our case, the positions of the wireless devices are the primary unknowns of interest.

In MCMC methods each instantiation of the network (i.e. with the variables having values) forms a state in a Markov chain. The sampling procedure generates a successive state with new values for some or all the variables. In effect, this process generates a new node in a Markov chain, where a node is an instance of the network. Eventually, the chain converges to a stationary distribution. Observing the values of the variables as the chain cycles along the stationary distribution allows the approximation of their true probability distribution. We call the process of drawing a random sample for all the variables in the network an *iteration*.

Although the MCMC methods proposed in [11] are both computationally efficient and provide quick convergence, they can still take a lot of time when many devices are localized simultaneously. For instance, they can take more than half a minute on a well-equipped machine to simultaneously localize 200 devices. We are thus motivated to explore parallel computing methods for this problem.

In this paper we describe two parallelization strategies. The first, inter-chain parallelism, runs multiple independent chains on different processors. The observed values are then aggregated to form the probability distributions of the variables. The second approach, intra-chain parallelism, divides the workload of a single iteration across processors.

We implemented our two approaches in Berkeley Unified Parallel C (BUPC) [19], which is a parallel language that adopts a Single Program Multiple Data (SPMD) model using a global address space (GAS). Specifically, we applied these two approaches to the most efficient MCMC inference method described in [11], which is called "whole domain sampling". We found UPC an effective language for describing the data layout needed by our algorithms.

We evaluated our implementation on three platforms: a 16-node symmetric multiprocessor (SMP), a 4-node cluster comprising of quad processors, and a 16 single-processor-node cluster. Our results show that intra-chain parallelism gives speedups of 12 on 16 processors on the first two platforms, when the MCMC method has performed a small number of iterations (at most 10,000). On the other hand, inter-chain parallelism requires many more iterations (at

least 40,000) on these two platforms in order to achieve speedups of 12 and higher. We found the 16-way cluster, which could only run the inter-chain algorithm, required at least 60,000 iterations to give a speedup of 12.

For the Bayesian networks we study here, it was shown in [11] that only a small number of iterations (at most 10,000) is required in order to get good localization results. Hence, intra-chain parallelism is a good candidate for applying parallelism to them, when run on platforms such as the first two. However, load balancing is harder to achieve in this partitioning scheme, since it requires splitting evenly the computational cost of a single iteration across the processors, which can be non-trivial. Additionally, it necessitates communication for every iteration. In inter-chain parallelism it is easier to achieve load balancing, since it only requires dividing evenly the number of iterations of the MCMC method among processors. For Bayesian networks that need many iterations in order to give good results, inter-chain parallelism is the algorithm to choose, because for a large number of iterations it outperforms intra-chain parallelism.

In order to analyze and predict the performance of our two algorithms we use the LogP [4] model and its extension for large messages, LogGP [1]. The predictions of the models are compared to the experimentally gathered data from the first two platforms. The comparisons show that the predictions are within 5% of the observed execution time for the inter-chain parallelism, whereas for intra-chain they are 7%-25% less than the actual time. The reason for the latter discrepancy is that there is load imbalance in the intra-chain parallelism that the models fail to capture. Nevertheless, the models can give us a good indication of the performance of our algorithms on different platforms.

The rest of this paper is organized as follows. We first present related work in Section 2. In Section 3 we give a brief description of the Bayesian networks we study in this work, while in Section 4 we describe the two parallel algorithms applied to the networks. In Section 5 we present our speedup results on different platforms and in Section 6 we analyze the performance of our algorithms using the LogGP model. Finally, Section 7 concludes the paper.

2 Related work

There are many active research efforts developing localization systems for wireless and sensor networks. In general, RSS-based localization systems have been shown to have average accuracies of 6-15ft depending on the level of training data used in a specific environment (for instance, a specific building floor [2, 3, 8]). The key advantage of RSS approaches is that they can use the existing packet traffic to localize. The absolute accuracies of the Bayesian networks explored here are on the higher end of these systems, with average accuracies of 15ft [15], but require much less training data when compared to other approaches. Fast MCMC-based solvers were implemented in [11] for the networks we consider.

Several researchers have proposed parallel algorithms for Bayesian inference. Specifically, [16] describes an algorithm that assigns variables to processors which can communicate directly only with the processors for "nearby" variables, as determined by the connections present in the Bayesian network. We believe that these kind of algorithms lack robustness, as they usually do not map with equal efficiency onto interconnection structures different from those for which they were designed. Moreover, they require a lot of synchronization and communication between processors.

Also, [9, 13, 14, 18] propose parallel algorithms, but they consider parallelizing only independent operations. Specifically, [9] describes parallel algorithms and their MPI-based implementation for Bayesian phylogenetic inference using MCMC, which are evaluated on a 32-node Beowulf cluster. In this work, processors are arranged in a 2D grid topology so that both chain-level and subsequence-level parallelism can be used. Furthermore, [13, 14] describe experimental results for a parallel version of a junction tree algorithm, implemented on a Stanford DASH multi-processor and a SGI Challenge XL. The algorithm transforms a Bayesian network into cliques and exploits parallelism across cliques (topological parallelism) and in cliques. Finally, [18] uses an algorithm by [17], which is a revision of [16] mentioned earlier, to map Bayesian networks onto hypercube parallel architectures. The mapping scheme maintains parent-child adjacency, is implemented and verified on a 64-node nCUBE. However, the algorithm has the same drawbacks as the one in [16]. In none of the approaches mentioned earlier some model of parallel computation (such as LogP/LogGP [1, 4]) was used to analyze the behavior of the algorithms as we do.

3 Localization networks

The Bayesian localization networks we consider here are called M_1, M_2, M_3, A_1 [7, 15]. They are acyclic digraphs (ADGs), whose vertices correspond to random variables and directed edges to dependencies among them. The networks are essentially a compact way to represent the joint probability distribution of all variables, whose value might be known (for instance, they might correspond to observations) or unknown. All models reflect the fact that signal strength decays approximately linearly with log distance. Some important variables of the networks are X, Y which represent the x- and y-coordinate of a location, b_{ij}, τ_i which are the parameters of the linear regression model that describes how a signal degrades linearly with log distance. In A_1 there are m signal strength readings at a particular lo-

cation (X, Y) with respect to the ith access point, and the signal is received by a mobile at an angle α_{ij}. An access point is a device whose location is known. The ratio $360/m$ is called granularity and tells us at what angle intervals the signal strengths are measured. A *training* set is one where we have signal strengths at known locations, while a *testing* set is a set of signal strengths at unknown locations. The training and testing sets are combined to form a complete ADG; the values in the training set affect the probability distribution of the position variables to be inferred from the testing set. More details about these models can be found in [7, 15].

4 Parallel algorithms

An inference method that can be used to estimate the values of variables in a Bayesian network is Markov Chain Monte Carlo (MCMC) simulation. It starts by assigning some initial value to each variable v with unknown value (e.g. x-coordinate of an object to be localized) and then cycles through the network replacing the old value of each v with a new one. Thus, the process consists of a number of iterations and during each iteration the old values of all variables are replaced with new ones. In the MCMC framework, the values are usually referred to as samples. After sufficient iterations of the procedure one assumes the Markov chain has reached its stationary distribution. We refer to the iterations before the chain has reached its stationary distribution as *burnin* and the ones after as *additional*. Below, we describe two parallel algorithms we apply to the MCMC process that infers values for the unknown quantities of the networks mentioned in Section 3. Both algorithms apply parallelism to the generation of a single Markov chain that requires B burnin iterations and A additional, and hence, the length of the chain is $B + A$.

4.1 Inter-chain parallelism

Inter-chain parallelism divides equally the additional iterations of the chain among all Q processors that are available. In the case of a P-way SMP, $Q = P$, whereas in a cluster of R P-way machines, $Q = P * R$. Thus, it runs Q chains in parallel each one with a different starting seed for the Markov walk. Different seeds ensure that each Markov walk will follow a different trajectory. Moreover, every chain needs to have B burnin iterations in order to ensure that the values chosen in the additional iterations are from the stationary distribution. Hence, the length of each chain is $B + A/Q$. Figure 1(a) shows a pictorial representation of how the algorithm divides the computational cost. Each row corresponds to a single iteration during which an MCMC method updates variables v_1, v_2, ... ,v_k, whereas each column corresponds to the sequence of values generated by the method for some variable after a number of iterations have been performed. Essentially, the algorithm "slices" horizontally the number of additional iterations, forcing though each chain to execute B burnin iterations.

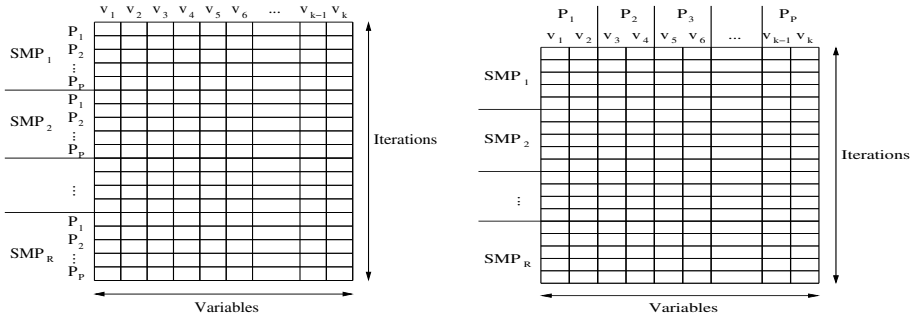
In order to generate statistics such as the median or the 95% interval, the samples of all variables generated in the additional iterations need to be sorted. There are several options that this can be done; through the use of some parallel sort (e.g. radix sort [6]) or processors can exchange the samples of the variables they control so that all samples of a given variable are collected by a single processor that can in turn sort them locally. After testing the performance of the two options, we have decided that the latter is faster, as the number of samples to be sorted per processor for our networks do not justify the use of a parallel sort. So, if there are k variables in a Bayesian network, processors divide them equally among them and each one sorts k/Q variables. Once sorting is done, statistics are gathered by a single processor that outputs the results to a file.

The algorithm manages to easily balance the sampling load on all processors, as each processor updates the same number of variables and generates chains of the same length. Also, running the algorithm on a cluster of SMPs is trivial, since all processors on the cluster are treated equally. The disadvantage of the algorithm though is that all processors need to pay an overhead of B burnin iterations.

4.2 Intra-chain parallelism

Intra-chain parallelism ensures that each SMP generates only one Markov chain regardless of the number of processors in it. Within a SMP the algorithm distributes the variables to be updated to the processors of the SMP, and, therefore, each processor updates only a subset of the variables of the Bayesian network. At the end of each iteration, every processor gathers the new values of the variables generated by the other processors of the SMP. The reason is that a processor requires the values of other variables in the network during the update process, and in order for the Markov chain to evolve, it is important that each processor has the latest value of the other variables.

For a single SMP, the algorithm generates only one chain of length $B + A$. For a cluster of R SMPs, the additional iterations are divided equally among the SMPs, so that there are R Markov chains that run in parallel, each of length $B + A/R$. Every SMP uses a different seed to evolve the Markov walk. The assignment of variables to processors is identical on all SMPs. Figure 1(b) depicts how the algorithm divides the computational cost among the SMPs. In particular, it "slices" the cost vertically within a SMP and horizontally across the SMPs of a cluster. It is interesting to note that in the case of a cluster of single-processor SMPs the intra-chain algorithm becomes inter-chain. There is no



(a) Inter-chain parallelism (b) Intra-chain parallelism

Figure 1. Sampling load distribution by our two parallel algorithms.

vertical "slicing" of the computational cost within a SMP; the single processor of a SMP updates all variables of the Bayesian network.

To generate statistics, the samples of all processors need to be sorted. Applying a parallel sort (e.g. radix sort) is harder when compared to inter-chain parallelism, as parallel sorts assume that all processors have samples for all variables. For this reason as well as the reasons mentioned in the previous section (4.1), we have processors sort samples locally in the cases of a single SMP and a cluster. Once statistics have been estimated, they are collected by a single processor that outputs them to a file.

Unlike inter-chain parallelism, the cost of the B burnin iterations is paid only once within each SMP. On the other hand, the algorithm necessitates an all-to-all exchange of values within a SMP at the end of each iteration. Furthermore, balancing the sampling load on the processors of a given SMP can be a non-trivial task, as the computational cost of updating a variable varies among the variables of a Bayesian network. The computational cost depends on the sampling method (e.g. slice sampling, conjugate sampling [10]) used to update a variable, as well as on the size of the training data given as input to the Bayesian network. A load imbalance will affect processors when they communicate at the end of each iteration, as some processors might have to wait for others to finish updating their variables.

Special consideration had to be taken when we implemented the intra-chain parallelism for our Bayesian networks. Assigning random subsets of the variables to different processors resulted in localization results that deviated from the results of a chain generated by a single processor that updates all variables together. The reason is that the MCMC method we use for inference and we call "whole domain sampling" [11] is a Gibbs sampling method. Applying parallelism to such a method by distributing variables to different processors requires an algorithm like the one described in [16]. As was explained in Section 2, these kind of algorithms do not map efficiently onto different interconnection structures, and thus we follow the vertical "slicing" described earlier. Nevertheless, we realized that the intra-chain algorithm can give results similar to a single chain, when applied to our Bayesian networks, by ensuring that certain groups of variables are assigned to the same processor. Specifically, variables b_{i0}, b_{i1} for the same i in M_1, M_2, and b_{i0}, b_{i1}, b_{i2}, b_{i3} for the same i in M_3, A_1, and X, Y of a specific location have to be on the same processor.

5 Performance results

We have implemented the two algorithms described in Section 4 using the Berkeley UPC (BUPC) [19] parallel language, which is an extension of C and provides a Global Address Space (GAS) model. In our work we used the 2.4.0 version of the BUPC compiler.

The algorithms were tested on three platforms. The first is a SMP with 16 processors running Linux. Each processor has a 2.4-GHz clock speed and 2 GBs of memory. The second is a cluster of 4 Pentium 3 machines, each one having a quad processor and running Linux. Processors have a 550-MHz clock speed and 250 MBs memory each. The nodes of the cluster are connected by a 100-Mbp switch. The third is a cluster of 16 Pentium 4 machines with a dual processor, running Linux and connected by a 100-Mbp switch. Processors have a 3.2-GHz clock speed and 500 MBs memory each. In all our experiments we used only one of the two processors of every node in the latter cluster.

All our Bayesian networks use training data in the learning process that maps signals to locations (M_1, M_2, M_3, A_1) and also to angles (A_1). For M_1, M_2, M_3 we used the BR dataset from [15] that contains 253 training points, was collected in a building that measures 255ft $\times$ 144ft and has 5 access points. For A_1 we used a dataset from [7] consisting of 20 training points collected in a building that measures 200ft $\times$ 80ft and has 4 access points. We followed the leave-n-out method, meaning that n points were chosen from the training set to be localized.

The results shown next were generated by applying both algorithms to a specific MCMC inference method, "whole domain sampling", that was shown in [11] to be the fastest in terms of time and convergence for the networks that we

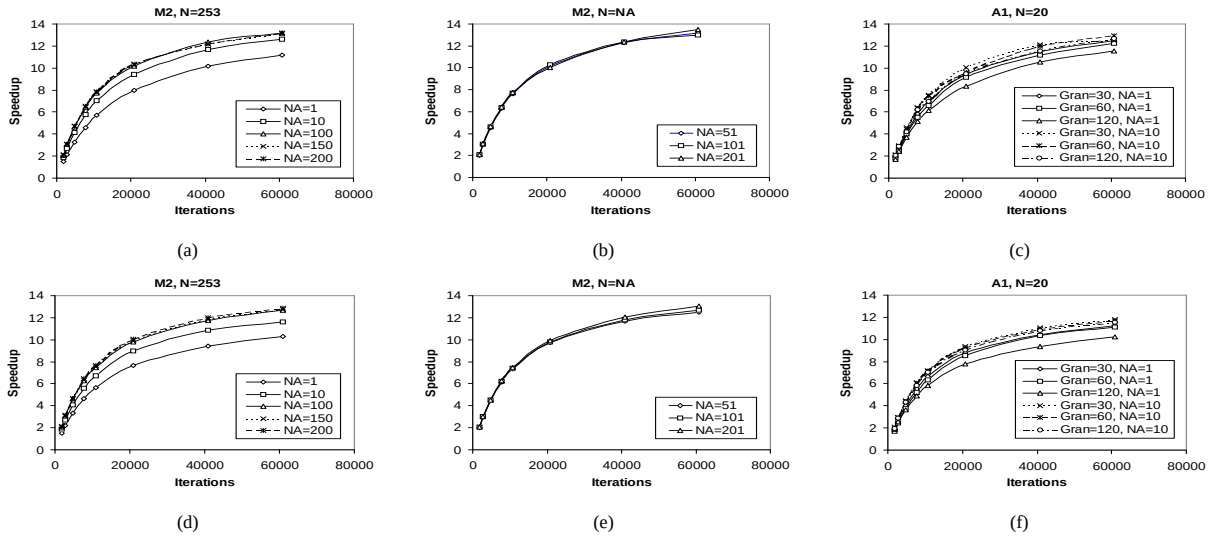


Figure 2. Speedups of the inter-chain parallelism on a 16-node SMP (a), (b), (c) and on a cluster of 4 quad-processor machines (d), (e), (f) with 16 threads (one per processor).

study here. However, they can be applied to other MCMC methods too. For the training data and the MCMC method we use, we have figured out that we need 800 burnin iterations. All our results are the average of 30 runs. In the graphs we describe next, N is the number of training points out of which we localize NA points. Also, $Gran$ is the granularity (see Section 3) used to take signal strength measurements in the A_1 network.

We evaluate the algorithms using the traditional speedup metric, defined as $Speedup = \frac{T_{serial}}{T_{parallel}(P)}$, where T_{serial} is the running time of the serial algorithm on one processor, and $T_{parallel}(P)$ is the running time of the parallel version of the algorithm on P processors. We use the time needed by a single-threaded BUPC program as T_{serial}, since we have managed to make it run as fast as the single-threaded C solvers proposed in [11]. There are different possibilities for measuring time in these cases, such as user time, wall clock time. In this paper we have chosen to measure wall clock time, that is the elapsed time between the start and the end of a run.

Finally, since speedups depend on the amount of computational cost that is distributed to processors, we present results with increasing values of NA and decreasing of $Gran$. This reveals the benefits we get from the two types of parallelism, when we localize many devices at the same time and we have many signal strength readings.

5.1 Inter-chain results

Figure 2 shows the speedups we get for some of our Bayesian networks with 16 threads on the 16-node SMP and the 4-machine cluster. The number of iterations in the graphs is the total number of iterations run by all threads together. As can be seen, both platforms give approximately the same speedup with the SMP offering slightly higher. The reason for the small improvement is that, in the cluster, threads need to use the Ethernet network to exchange the variable samples in order to sort them and produce statistics. Our results show that inter-chain parallelism does not offer good speedups as the number of iterations decrease. The reason is that, although the computational cost of the additional iterations is divided among all threads, the cost of the burnin is not; it is a fixed overhead that all threads need to pay. We see the algorithm needs at least 40,000 iterations in order to pay off the burnin computational cost of 800 iterations and give speedups of 12. However, as was shown in [11], for our networks we do not need more than 10,000 iterations to get good localization results. The results for the other networks (M_1, M_3) are similar to the ones shown in graphs 2(a), 2(d) (see [12]).

5.2 Intra-chain results

Figure 3 depicts the speedups of the intra-chain parallelism when running 16 threads on the 16-node SMP and the 4-machine cluster. We see that on the 16-node SMP speedups are very good even when the number of iterations scale down to 2,000. In the cluster on the other hand, speedups are low for a small number of iterations (2,000 to 5,000), but after that they reach the level of speedups offered by the 16-node SMP. The reason is that on the 16-node SMP there is only one Markov chain and hence the cost of the burnin iterations is paid only once. In the case of the cluster, since it consists of 4 machines, the algorithm ran 4 chains in parallel and each one had to pay the overhead of

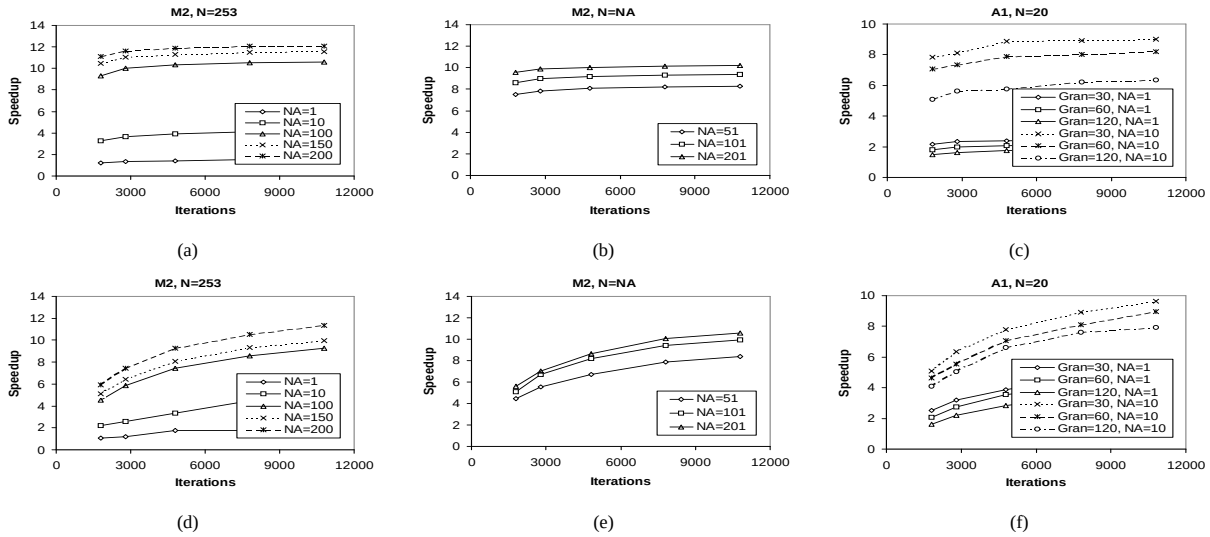


Figure 3. Speedups of the intra-chain parallelism on a 16-node SMP (a), (b), (c) and on a cluster of 4 quad-processor machines (d), (e), (f) with 16 threads (one per processor).

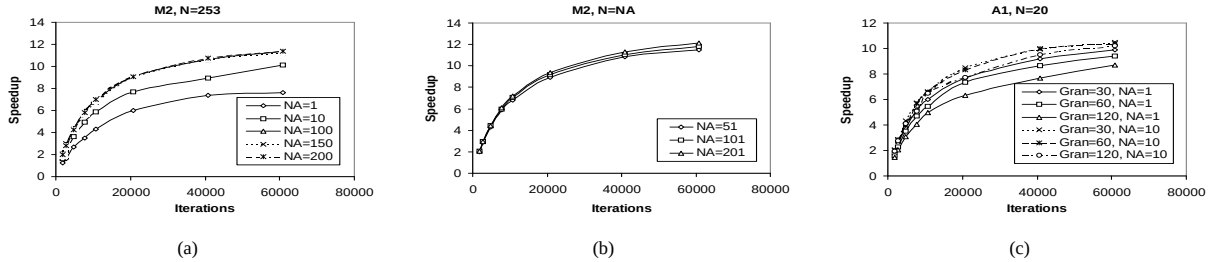


Figure 4. Speedups of the intra-chain parallelism on a cluster of 16 machines with one thread per machine. The algorithm is essentially inter-chain on the cluster.

the burnin. The graphs show that after 5,000 iterations the algorithm pays off the burnin cost giving higher speedups.

An important issue in this algorithm is balancing the load within a SMP. This requires assigning variables to the processors in such a way so that in every iteration the total time spent on each processor for updating variables is roughly equal. We used the computational cost required to update each variable of the network to assign variables to processors at compile time under the restrictions explained in Section 4.2. Since load balancing can be achieved more easily on 4 processors (each machine in the cluster is a quad Pentium 3) rather than on 16, when the number of iterations increase, the cluster can achieve better speedups than the 16-node SMP. This can be clearly seen for M_2 when $N = NA = 201$ (graphs 3(b), 3(e)), and for A_1 when $Gran = 30$, $NA = 10$ (graphs 3(c), 3(f)) for 10,000 iterations, where the cluster starts outperforming the SMP.

Graph 3(b) shows that when localizing all the points in the training set ($N = NA$) we get speedups of up to 10, whereas in 3(a) the same network (M_2) can give better speedups (up to 12) when localizing $NA < N$ points. As was shown in [11], in the $NA = N$ case we need to bound the b_{i0} parameters for all i, resulting in a high computational cost when updating these variables. As a consequence, we were not able to achieve good load balancing in this case on the 16 processors of the SMP. Also, graphs 3(c), 3(f) show that we do not achieve as high speedups for A_1 as for the other networks. This is due to the fact that the training data for this network consists of only 20 points and hence there is not enough workload to distribute to the processors that will offset the communication required at the end of each iteration and thus achieve speedups comparable to the other networks. Networks M_1, M_3 have performance similar to the one shown in graphs 3(a), 3(d) (see [12]).

As was mentioned in Section 4.2, the algorithm behaves like the inter-chain when run on a cluster with 1 thread per SMP. Figure 4 presents its performance when run on our 16-node cluster. The curves are similar to the ones in Figure 2, but the speedups are smaller when compared to that figure. The reason for the decrease is that each thread on the 16-node cluster uses the Ethernet network to exchange samples with other threads, whereas in Figure 2 there is com-

LogP/LogGP		16-node SMP	Cluster SMP	Cluster Net
Latency	L	0.39870 μsec	0.51790 μsec	72.1308 μsec
Overhead	o	0.04315 μsec	0.25655 μsec	24.2436 μsec
Gap	g	0.10330 μsec	0.62310 μsec	50.2202 μsec
Gap-per-byte	G	0.00263 $\frac{\mu sec}{byte}$	0.00977 $\frac{\mu sec}{byte}$	0.33447 $\frac{\mu sec}{byte}$

Table 1. LogP/LogGP model parameters on a 16-node SMP and a cluster of 4 quad-processor machines.

munication within a SMP on the 4-machine cluster apart from across the Ethernet during the exchange, and on the 16-node SMP there is no usage of Ethernet at all. Figure 4 shows that for M_2 we need at least 60,000 iterations to get a speedup of 12, whereas for A_1 we need even more.

6 LogGP analysis

In this section we describe a way of modeling the behavior of our algorithms so that we understand how much time each phase of the algorithms requires as well as we can predict their performance on several platforms.

6.1 Modeling communication and computation

In order to describe the network performance of our algorithms we use the LogP [4] model, which is a well-established approach to modeling small messages, and its extension, LogGP [1], for large messages. To analyze the performance of our algorithms on a cluster, we use two types of LogGP parameters; one for communication within a SMP of the cluster and one for communication between SMPs. Table 1 summarizes the parameter values of the LogP/LogGP model on two platforms. The parameters were measured as in [5], by writing BUPC benchmark programs. The value of G shown in the table is the worst possible G measured from the benchmarks.

$$T_{Iter,M_1} = \sum_{j=1}^{2*d} t_{b_{i01}} + \sum_{j=1}^{d} t_{\tau_i} + \sum_{j=1}^{NA} t_X + \sum_{j=1}^{NA} t_Y \qquad (1)$$

Local computation is captured by measuring time per variable per processor. Formula 1 estimates the sampling time of one iteration in the M_1 Bayesian network. In this formula, d is the number of access points (see Section 3), and NA the number of points we try to localize. Also, $t_{b_{i01}}$ represents the time needed to update either variable b_{i0} or b_{i1}, while t_{τ_i}, t_X, t_Y the time needed for variables τ_i, X, Y respectively (see Section 3). Table 2 presents the local computation rates of M_1, A_1 for the 16-node SMP. When possible we use a constant value for the time per variable. However, in the case of $t_{b_{i01}}/t_{b_{i03}}$, t_{τ_i}, we use times per variable per training size (N) per number of points to localize (NA), because computation is dependent on these parameters. Additionally, there is dependency on granularity in A_1. The value of t_{gen_stats} corresponds to a measured time per sample needed to generate statistics. This includes the time of local quick sort, finding median, average, 2.5% and 97.5% interval, and standard deviation. Formulas like 1 and rates for the other networks can be found in [12].

Variable	$\mathbf{M_1}$	$\mathbf{A_1}$
$t_{b_{i01}}/t_{b_{i03}}$	0.00045*N*NA	0.00442*N*NA*360/Gran
t_{τ_i}	0.00044*N*NA	0.00455*N*NA*360/Gran
t_X	3.14835	20.4175
t_Y	2.61147	15.4654
t_{gen_stats}	0.13406	0.13468
t_α		0.57787

Table 2. Local computation rates (in μsecs) for the 16-node SMP.

Table 3 summarizes the time required by the algorithms on two platforms as this is estimated by the LogGP models. The two platforms are a P-way SMP and a cluster of R P-way machines. The formulas for $T_{intra,smp}$, $T_{intra,clu}$, $T_{inter,smp}$, $T_{inter,clu}$ capture the phases of the algorithms that take most of the time in the algorithms. Specifically, these phases are the sampling process, generation of statistics (t_{gen_stats}), moving samples to other processors so that they can be sorted ($t_{move\_samples}$), and gathering values from all processors within a SMP at the end of each iteration in the intra-chain parallelism (t_{gather_values}).

In the formulas we assume the algorithms try to parallelize the generation of a Markov chain with length $total\_iter$ that consists of $burnin$ burnin iterations and $additional\_iter$ additional. Furthermore, the number of variables assigned by the intra-chain algorithm to processors within an SMP as well as the number of variables sorted by each processor in the inter-chain algorithm are approximated by $network\_vars/P$, where $network\_vars$ is the total number of variables in the Bayesian network with unknown value. The approximation is good for the inter-chain parallelism as processors are assigned equal number of variables to sort (see Section 4.1). On the other hand, it is rougher for intra-chain, as processors are assigned at compile time different number of variables to update based on the computational cost required to generate a new sample for a variable. Additionally, since variables are represented in our implementation as floating point numbers, each sample of a variable requires $sizeof(float)$ bytes. All barriers in our implementation require $\lceil log_2(P) \rceil$ phases to complete and hence their running time is estimated as $\lceil log_2(P) \rceil * (L + o + g)$.

Finally, in order to distinguish the LogGP parameters that refer to communication within a SMP and communication using the network connecting SMPs, in Table 3 we use the L, o, g, G notation for the first case, and L_{net}, o_{net}, g_{net}, G_{net} for the latter.

Algorithm	Time
Intra-chain (SMP)	$T_{intra,smp} = \left(\frac{\text{Sampl. time of 1 iter}}{P} + t_{gather\_values}\right) * total\_iter + t_{gen\_stats} * additional\_iter * vars\_per\_processor$
	$t_{gather\_values} = (P-1) * (L + o + (m-1) * G + g) + 2 * (\lceil log_2(P) \rceil * (L + o + g))$
	$vars\_per\_processor \approx network\_vars/P$
	$m \approx vars\_per\_processor * sizeof(float)$
Intra-chain (cluster)	$T_{intra,clu} = \left(\frac{\text{Sampl. time of 1 iter}}{P} + t_{gather\_values}\right) * \left(burnin + \frac{additional\_iter}{R}\right) + t_{move\_samples} + t_{gen\_stats} * additional\_iter * \frac{vars\_per\_processor}{R}$
	$t_{gather\_values} = (P-1) * (L + o + (m-1) * G + g) + 2 * (\lceil log_2(P) \rceil * (L + o + g))$
	$t_{move\_samples} = (R-1) * (L_{net} + o_{net} + (n-1) * G_{net} + g_{net}) + 2 * (\lceil log_2(P*R) \rceil * (L_{net} + o_{net} + g_{net}))$
	$vars\_per\_processor \approx network\_vars/P$
	$m \approx vars\_per\_processor * sizeof(float)$
	$n \approx (vars\_per\_processor * additional\_iter * sizeof(float))/R^2$
Inter-chain (SMP)	$T_{inter,smp} = (\text{Sampl. time of 1 iter}) * \left(burnin + \frac{additional\_iter}{P}\right) + t_{move\_samples} + t_{gen\_stats} * additional\_iter * vars\_per\_processor$
	$t_{move\_samples} = (P-1) * (L + o + (m-1) * G + g) + 2 * (\lceil log_2(P) \rceil * (L + o + g))$
	$vars\_per\_processor \approx network\_vars/P$
	$m \approx (network\_vars * additional\_iter * sizeof(float))/P^2$
Inter-chain (cluster)	$T_{inter,clu} = (\text{Sampl. time of 1 iter}) * \left(burnin + \frac{additional\_iter}{P*R}\right) + t_{move\_samples} + t_{gen\_stats} * additional\_iter * vars\_per\_processor$
	$t_{move\_samples} = (P-1) * (L + o + (m-1) * G + g) + ((R-1) * P) * (L_{net} + o_{net} + (m-1) * G_{net} + g_{net}) + 2 * (\lceil log_2(P*R) \rceil * (L_{net} + o_{net} + g_{net}))$
	$vars\_per\_processor \approx network\_vars/(P*R)$
	$m \approx (network\_vars * additional\_iter * sizeof(float))/(P*R)^2$

Table 3. Time of each algorithm on two platforms.

6.2 Measured vs. predicted results

Figure 5 displays the total measured time of the inter-chain algorithm as well as its different phases, measured and predicted, for network M_2 on the 16-node SMP and the 4-machine cluster. Formulas $T_{inter,smp}$ and $T_{inter,clu}$ (Table 3) were used for the predicted time in graphs 5(b) and 5(d) respectively. We see that, on both platforms, the overall predicted time by our models closely matches the measured time, as it is within 5% of it. From the breakup of the total time into phases, it is obvious that sampling dominates execution time on both platforms; it accounts for at least 90% of the running time. The "move samples" phase, is the phase where processors exchange samples to be sorted. For the SMP this time is negligible, whereas for the cluster is considerably greater as the per-byte bandwidth (G in Table 1) is much larger on the network than within a SMP.

Figure 6 displays the total measured time of the intra-chain algorithm as well as its different phases, measured and predicted, for network M_2. An important phase of this algorithm is the "gather values" phase, during which processors within a SMP exchange the newly-generated values of the variables they control at the end of each iteration. The actual total time needed for this phase depends on how equally-distributed the sampling load is among the processors of a SMP. If the load is not equally balanced, then some processors will have to wait for others to finish updating their variables in order to receive these new values. We see from the graphs that the time this phase takes as a percentage of the total execution time can range from 7.8% (graph 6(f)), to 19% (graph 6(b)) and up to 25% (graph 6(d)).

When we used formulas $T_{intra,smp}$, $T_{intra,clu}$ (Table 3) to predict the running time of the algorithm, we saw that the estimated time given by the models was away from the measured; the difference ranged from 7% to 25%. There could be two reasons for this: (a) load imbalance, (b) contention. Both these factors are not captured in our models. We measured the load imbalance by subtracting the minimum time spent in the "gather values" phase from the maximum time spent in the phase among all processors within a SMP. The imbalance time was added as a percentage to the predicted time in graphs 6(b), 6(d), 6(f). The latter graphs show that now the total predicted time with the added imbalance is close to the actual running time, and imbalance ranges from 6.4% (graph 6(f)), to 10.6% (graph 6(b)) and up to 21.7% (graph 6(d)). In addition, they show that imbalance is higher on the 16-node SMP (graphs 6(b), 6(d)) than on the cluster (graph 6(f)). The reason is that the SMP has 16 processors whereas each SMP on the cluster has 4 and distributing the load evenly, under the restrictions explained in Section 4.2, can be done more easily on a 4-way machine than on a 16-way. The effect of not dividing the load evenly among processors is more intense for the M_2 network when $N = NA$ (graph 6(d)). As was explained in Section 5.2, for this case we had to bound the b_{i0} parameters (see Section 3) for all i, resulting in not good load balancing. Moreover,

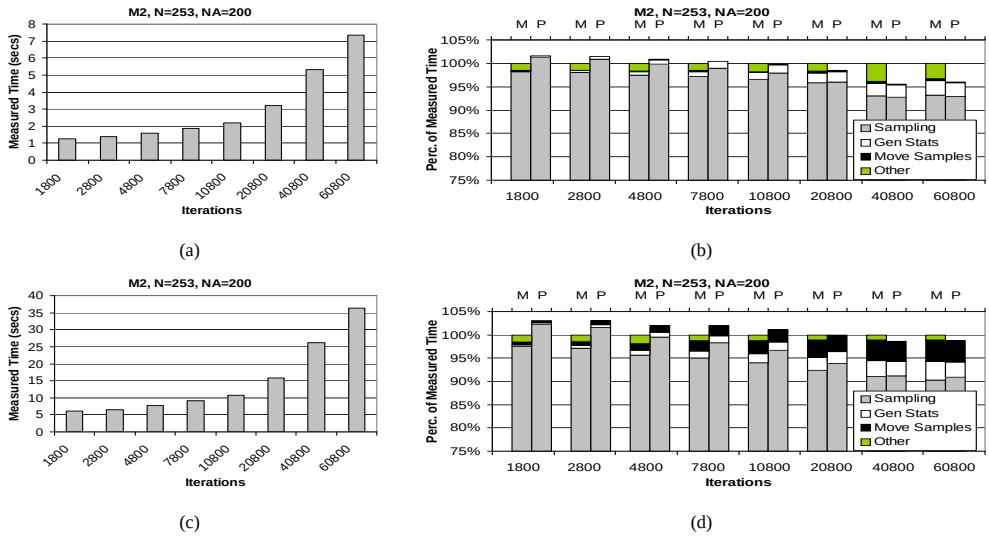


Figure 5. Performance of the inter-chain parallelism on a 16-node SMP (a), (b) and on a cluster of 4 quad-processor machines (c), (d) with 16 threads. "M" is for measured and "P" for predicted.

not having a good load balance affects the predicted sampling time, because in the $T_{intra,smp}$, $T_{intra,clu}$ formulas (Table 3) we divide the sampling cost of one iteration by P. In general, dividing by P underestimates the sampling cost of the intra-chain algorithm by a factor that depends on the load imbalance. For instance, when $NA < N$ in the M_2 network (graph 6(b), 10800 iterations), our model underestimates the sampling time by 3.1%, whereas in the $N = NA$ case (graph 6(d), 10800 iterations), the sampling time is underestimated by 8%.

The results for the other Bayesian networks (for both algorithms) are qualitatively similar and can be found in [12].

7 Conclusions

In this work we describe two parallel algorithms for MCMC-based Bayesian inference in indoor positioning systems. The algorithms apply inter-chain and intra-chain parallelism on the Markov chain generation, were implemented using BUPC, and tested on three platforms; a 16-node SMP, a 4-node cluster of quad processors, and a 16 single-processor-node cluster. Our results show that intra-chain parallelism achieves a speedup of 12 on the first two platforms with at most 10,000 iterations. This fact makes the algorithm particularly attractive for our positioning systems, since it was shown in [11] that they do not need more than 10,000 iterations to provide good localization results. On the other hand, inter-chain parallelism requires at least 40,000 iterations on the first two platforms and at least 60,000 iterations on the third in order to give speedups of 12 or more. Hence, the latter algorithm is suitable for applications that require long Markov chains.

We also use the LogP/LogGP model of parallel computation to analyze the behavior of these algorithms and predict their performance on different platforms. Our analysis shows that the predicted time of the model is within 5% of the measured for inter-chain parallelism, whereas for intra-chain it is 7%-25% less due to load imbalance that the algorithm suffers from.

8 Acknowledgements

We would like to thank Prof. Alan George and Adam Leko from UFL for letting us use a 16-processor SMP, as well as the Berkeley UPC group for their help. This work was supported in part by NSF grant CNS-0448062.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model. *Journal of Parallel and Distributed Computing*, 44:71–79, 1997.

[2] P. Bahl and V. N. Padmanabhan. RADAR: An In-Building RF-Based User Location and Tracking System. In *Proceedings of the 19th IEEE International Conference on Computer Communications (INFOCOM)*, March 2000.

[3] R. Battiti, M. Brunato, and A. Villani. Statistical Learning Theory for Location Fingerprinting in Wireless LANs. Technical Report DIT-02-086, University of Trento, Informatica e Telecomunicazioni, Oct. 2002.

[4] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

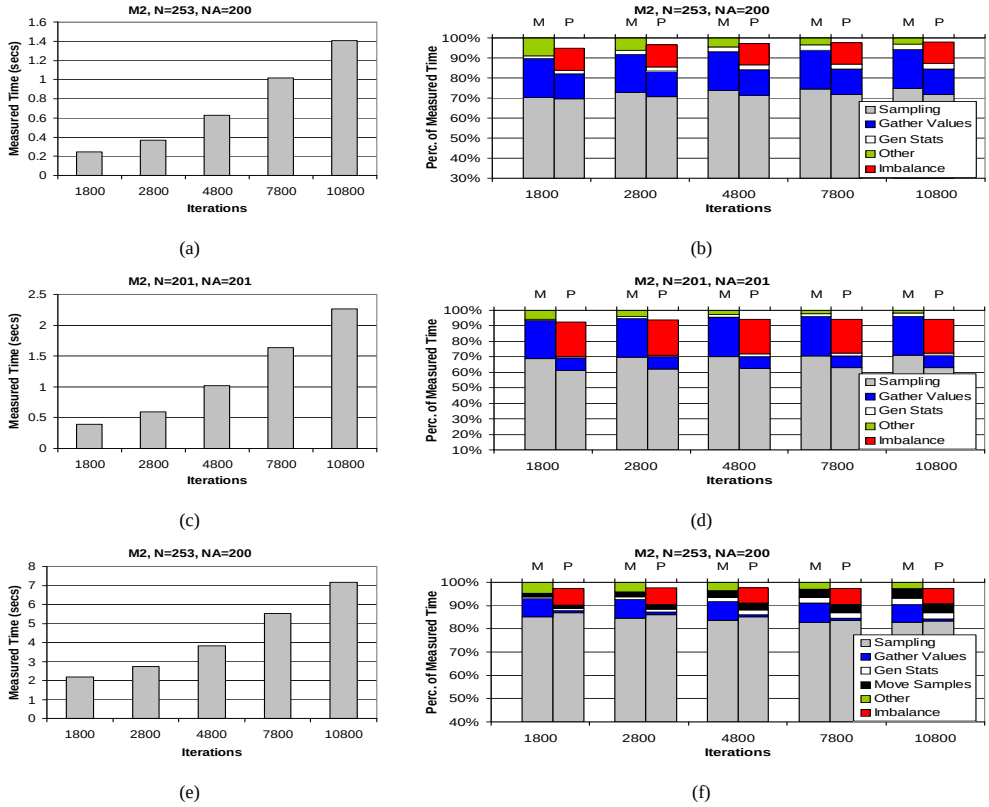


Figure 6. Performance of the intra-chain parallelism on a 16-node SMP (a)-(d) and on a cluster of 4 quad-processor machines (e), (f) with 16 threads. "M" is for measured and "P" for predicted.

[5] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. In *IEEE Micro*, Feb. 1996.

[6] A. C. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Technical Report UCB//CSD-94-829, University of California, Berkeley, Department of Electrical Engineering and Computer Science, 1994.

[7] E. Elnahrawy, J.-A. Francisco, and R. P. Martin. Adding Angle of Arrival Modality to Basic RSS Location Management Techniques. In *Proceedings of IEEE International Symposium on Wireless Pervasive Computing (ISWPC)*, Feb. 2007.

[8] E. Elnahrawy, X. Li, and R. P. Martin. The Limits of Localization Using Signal Strength: A Comparative Study. In *Proceedings of the First IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON)*, Oct. 2004.

[9] X. Feng, D. A. Buell, J. R. Rose, and P. J. Waddell. Parallel Algorithms for Bayesian Phylogenetic Inference. *Journal of Parallel and Distributed Computing*, 63:707–718, 2003.

[10] W. R. Gilks, S. Richardson, and D. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman and Hall, London, 1996.

[11] K. Kleisouris and R. P. Martin. Reducing the Computational Cost of Bayesian Indoor Positioning Systems. In *Proceedings of the Third IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON)*, Sept. 2006.

[12] K. Kleisouris and R. P. Martin. Parallel Algorithms for Bayesian Indoor Positioning Systems. Technical Report DCS-TR-612, Department of Computer Science, Rutgers, The State University of New Jersey, Piscataway, New Jersey, March 2007.

[13] A. V. Kozlov and J. P. Singh. A Parallel Lauritzen-Spiegelhalter Algorithm for Probabilistic Inference. In *Proceedings of Supercomputing*, Nov. 1994.

[14] A. V. Kozlov and J. P. Singh. Parallel Implementations of Probabilistic Inference. *Computer*, 29:33–40, 1996.

[15] D. Madigan, E. Elnahrawy, R. Martin, W. Ju, P. Krishnan, and A. Krishnakumar. Bayesian Indoor Positioning Systems. In *Proceedings of the 24th IEEE International Conference on Computer Communications (INFOCOM)*, pages 324–331, March 2005.

[16] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, California, 1988.

[17] M. A. Peot and R. D. Shachter. Fusion and Propagation with Multiple Observations in Belief Networks (Research Note). *Artificial Intelligence*, 48:299–318, 1991.

[18] N. Saxena, S. Sarkar, and N. Ranganathan. Mapping and Parallel Implementation of Bayesian Belief Networks. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, 1996.

[19] The Berkeley UPC Compiler, 2002. http://upc.lbl.gov.