2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

A parallel EM algorithm for Gaussian Mixture Models implemented on a NUMA system using OpenMP

Wojciech Kwedlo
Faculty of Computer Science
Białystok University of Technology
Wiejska 45a, 15-351 Białystok, Poland
w.kwedlo@pb.edu.pl

***Abstract*—In the paper the problem of estimation of Gaussian mixture model parameters is considered. A shared memory parallelization of the standard EM algorithm, based on data decomposition, is proposed. Our approach uses a rowwise block striped decomposition of large arrays storing feature vectors and posterior probabilities. Additionally, some NUMA optimizations, which allow threads to use as much local memory as possible, exploiting the *first-touch* memory allocation policy of the Linux operating system, are described. The proposed method was implemented in OpenMP and tested on a 64-core system based on four AMD Opteron 6272 (codenamed "Interlagos") processors. The experimental results indicate, that on large datasets, the algorithm scales very well with respect to the number of cores, and NUMA optimizations significantly improve its performance.**

***Keywords*—Gaussian mixture model; EM algorithm; OpenMP; NUMA**

I. Introduction

Gaussian Mixture Model (GMM) is a probability density function, which can be expressed as a weighted sum of multivariate Gaussian components. GMMs, which can effectively model complex probability distributions, are widely used in such applications as data clustering [1], [2], data classification [3], speaker recognition [4], or image segmentation and classification [5]. The standard method for estimation of GMM parameters is the expectation-maximization (EM) algorithm [6]. Unfortunately, the EM learning of GMM parameters is a computationally demanding task. For that reason the serial version of EM is unable to process large datasets consisting of millions of feature vectors. Due to this limitation various parallel versions have been proposed. According to our knowledge, the first parallel version of EM, implemented using MPI, was described in [7]. A more efficient parallel implementation for multicore cluster was proposed in [8]. In [9] an implementation of the EM using the FREERIDE middleware, which supports both distributed and shared memory parallelization, was proposed. A parallel version of EM implemented for NVIDIA's GPUs in CUDA environment was described in [10].

With the advent of relatively cheap multicore NUMA machines from Intel (maximum 80 cores using 8 Xeon E7-8000 series CPUs) and AMD (maximum 64 cores using 4 Opteron 6200/6300 series CPUs) it is interesting to investigate parallelization of the EM algorithm in such environments. In such systems cores and memory modules are connected via a cache-coherent network. Each core has access to any memory location but access to local memory (i.e., located in the same network node) is much faster than access to remote memory (i.e., located in other network nodes). A crucial requirement for achieving high performance in NUMA systems is such placement of data and computation that data needed by each thread are local to the core on which the thread is running. The main contribution of this paper is a parallel version of the EM for GMMs, implemented using OpenMP [11], optimized for such systems. Our version effectively exploits the *first-touch* policy of the Linux operating system [12]. This policy allocates a memory page on a node of the first core which accesses the page. Our experiments show, that by using this property to make most memory accesses local, we can significantly improve the performance of the parallel EM algorithm.

The rest of the paper is organized as follows. In the next section some background knowledge about GMMs learning is presented. In section III EM algorithm for GMMs is outlined. Section IV presents the hardware architecture of our multicore system. The method of parallelization of the EM algorithm is presented in section V. Section VI presents the results of experimental scalability study of our approach. The last section concludes the paper.

II. Background on GMMs

A finite mixture model $p(\mathbf{x}, \Theta)$ can be expressed by a weighted sum of $K > 1$ components:

$$p(\mathbf{x}|\Theta) = \sum_{m=1}^{K} \alpha_m p_m(\mathbf{x}|\theta_m), \tag{1}$$

where α_m is mth mixing proportion and p_m is the probability density function of the mth component. In (1) θ_m is the set of parameters defining the mth component and $\Theta = \{\theta_1, \theta_2, \ldots, \theta_K, \alpha_1, \alpha_2, \ldots, \alpha_K\}$ is the complete set of the parameters needed to define the mixture. The functional form of p_m is assumed to be known; Θ is unknown and has to be estimated. The mixing proportions must satisfy the following conditions: $\alpha_m > 0, \quad m = 1, \ldots, K$ and $\sum_{m=1}^{K} \alpha_m = 1$. The number of components K is either known a priori or has to be determined during the mixture learning process. In the paper we assume, that K is known.

1066-6192/14 $31.00 © 2014 IEEE
DOI 10.1109/PDP.2014.77

In GMMs the probability density function of the mth component is given by:

$$p_m(\mathbf{x}|\theta_m) = \frac{1}{(2\pi)^{d/2}|\Sigma_m|^{1/2}} \exp(-\frac{1}{2}(\mathbf{x}-\mu_m)^T\Sigma_m^{-1}(\mathbf{x}-\mu_m)), \quad (2)$$

where μ_m and Σ_m denote the mean vector and covariance matrix, respectively, $|\cdot|$ denotes a determinant of a matrix, ^{T} denotes transposition of a matrix, and d is the dimension of the feature space. The set parameters of the mth component is $\theta_m = \{\mu_m, \Sigma_m\}$. Thus, for the GMM Θ is defined by: $\Theta = \{\mu_1, \Sigma_1, \ldots, \mu_K, \Sigma_K, \alpha_1, \ldots, \alpha_K\}$.

Estimation of the parameters of a GMM can be performed using the maximum likelihood approach. Given a training set of independent and identically distributed feature vectors $X = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\}$, where $\mathbf{x}^i = [x_1^i, x_2^i, \ldots, x_d^i] \in R^d$, the loglikelihood corresponding to the K-component GMM is given by:

$$\log p(X|\Theta) = \log \prod_{i=1}^{N} p(\mathbf{x}^i|\Theta) = \sum_{i=1}^{N} \log \sum_{m=1}^{K} \alpha_m p_m(\mathbf{x}^i|\theta_m). \quad (3)$$

The maximum likelihood estimate of the parameters is given by: $\Theta_{ML} = \text{argmax}_\Theta\{\log p(X|\Theta)\}$. It is well known that the solution of this maximization problem cannot be obtained in closed form (e.g. [13]). Thus, a numerical optimization method has to be employed to find it.

III. EM Algorithm for Gaussian Mixture Models

The most popular approach for maximizing (3) is the EM algorithm. It is an iterative algorithm, which, starting from initial guess of a parameters $\Theta^{(0)}$, generates a sequence of estimations $\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(j)}, \ldots$, with increasing loglikelihood (i.e., $\log p(X|\Theta^{(j)}) > \log p(X|\Theta^{(j-1)})$). Each iteration j of the algorithm consists of two steps called expectation step (E-step) and maximization step (M-step) followed by a convergence check. For the GMMs these steps are defined as follows [6]:

1) E-step: Given the set of mixture parameters $\Theta^{(j-1)}$ from the previous iteration, for each $m = 1, \ldots, K$ and $i = 1, \ldots, N$, the posterior probability that a feature vector $\mathbf{x}^i$ was generated from mth component is computed as:

$$h_m^{(j)}(\mathbf{x}^i) = \frac{\alpha_m^{(j)} p_m(\mathbf{x}^i|\theta_m^{(j-1)})}{\sum_{k=1}^{K} \alpha_k^{(j)} p_k(\mathbf{x}^i|\theta_k^{(j-1)})}, \quad (4)$$

where $\theta_m^{(j-1)}$ and $\theta_k^{(j-1)}$ denote parameters of components m and k, in the iteration $j-1$, respectively.

2) M-step: Given the posterior probabilities $h_m^{(j)}(\mathbf{x}^i)$ obtained in the E-step the set of parameters $\Theta^{(j)}$ is calculated as:

$$\alpha_m^{(j)} = \frac{1}{N} \sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i) \quad (5)$$

$$\mu_m^{(j)} = \frac{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i) * \mathbf{x}^i}{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)} \quad (6)$$

$$\Sigma_m^{(j)} = \frac{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)(\mathbf{x}^i - \mu_m^{(j)})(\mathbf{x}^i - \mu_m^{(j)})^T}{\sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)} \quad (7)$$

3) Convergence check: The loglikelihood $\log p(X|\Theta^{(j)})$ is computed according (3). The algorithm is terminated if the following convergence criterion is met.

$$\frac{\log p(X|\Theta^{(j)}) - \log p(X|\Theta^{(j-1)})}{\log p(X|\Theta^{(j)})} < \epsilon, \quad (8)$$

where $\epsilon \ll 1$ is a user defined termination threshold. If the convergence criterion is not met algorithm proceeds to Step 1.

IV. Hardware Architecture

The NUMA system used in this study was a Dell Power Edge R815 equipped with four Opteron 6272 (codenamed "Interlagos") processors. This processor is based on a Bulldozer module [14], which represents departure from traditional x86 design practices. The traditional x86 designs share memory controllers, IO interfaces and optionally last level of cache among multiple cores in a single chip. Contrary to this, Bulldozer shares substantial resources among two cores in a module. Fig. 1 shows, that each module contains two separate integer cores with full out-of-order architecture. However, the floating point unit is shared between two cores in a module. This unit is equipped with two 128-bit FMAC (fused multiply accumulate) engines, which in a single clock cycle can issue two 128-bit floating point SSE instructions to either of cores or one 128-bit instruction to each of cores. In an alternate mode of operation two 128-bit engines can be combined together to issue one 256-bit AVX instruction to either of cores. The above scheme for sharing floating points units allows limiting the power consumption of the processor at the cost of introducing resource contention. It is interesting to notice, that if the second core is idle, the first one can operate with twice floating point throughput (two 128-bit instructions issued per clock cycle).

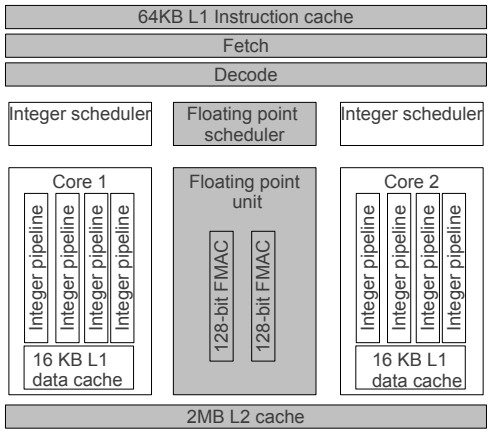

Fig. 1. Simplified architecture of a Bulldozer module. The components shared among two cores are grayed.

Fig. 2 shows, that the "Interlagos" processor is a multi chip module (MCM) consisting of two dies connected using one

full width (16 bits) and one half width (8 bits) Hypertransport links. Each die consists of four Bulldozer modules (giving total eight cores), a two channel DRAM controller, and 8 MB L3 cache. Thus, the processor is inherently a NUMA architecture with two nodes. From the view point of the Linux operating system a Dell Power Edge R815 server equipped with four "Interlagos" processors is a NUMA machine with 8 nodes of 8 cores each. The nodes are connected by cache coherent Hypertransport links. Fig. 3 shows the interconnection network topology [15]. The diameter of the interconnection network i.e., the maximal number of hops between the NUMA nodes is equal 2. Our system is equipped with 256 GB of RAM distributed evenly between NUMA nodes (32 GB per node).

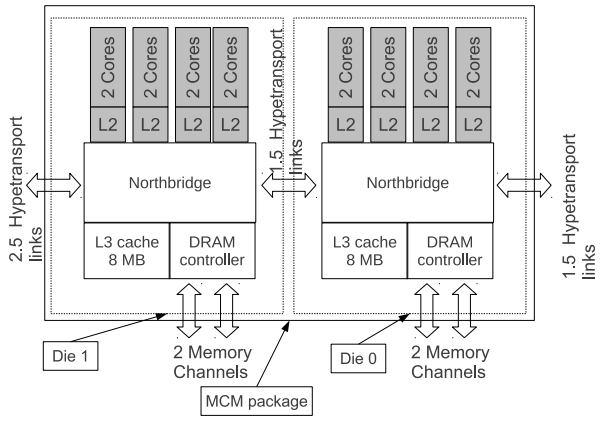

Fig. 2. Architecture of the Opteron 6272 codename "Interlagos" processor.

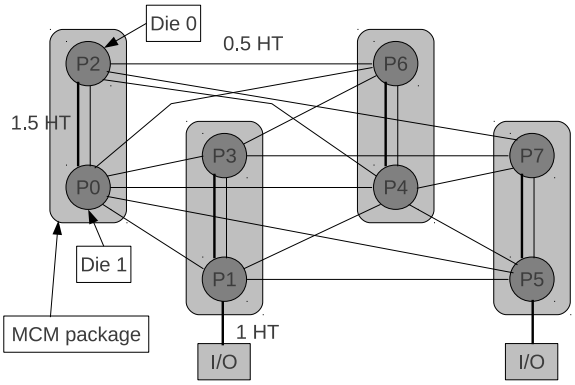

Fig. 3. Simplified topology (memory channels are not shown) of the NUMA system used in the experiments.

V. Parallelization using OpenMP

A. Problem decomposition

In each iteration the algorithm operates on two large arrays: the array storing elements x_j^i (where $i = 1, \ldots, N$ and $j = 1, \ldots, d$) of each feature vector and the array storing posterior probabilities $h_m^{(j)}(\mathbf{x}^i)$ (where $i = 1, \ldots, N$ and $m = 1, \ldots, K$). The former is used as input throughout the run of the EM algorithm, whereas the latter is the output of the E-step and serves as an input in M-step. Fig. 4 shows, that both arrays are partitioned evenly into threads using the rowwise block striped decomposition. Based on these partitions, the loops iterating over the N rows of two arrays can be parallelized. The parallelization is performed by using OpenMP `omp for` directive with static scheduling of loop iterations [11], which guarantees the assignment of data instances to threads shown in Fig. 4

The other data used by algorithm are the model parameters $\Theta^{(j)} = \{\mu_1^{(j)}, \Sigma_1^{(j)}, \ldots, \mu_K^{(j)}, \Sigma_K^{(j)}, \alpha_1^{(j)}, \ldots, \alpha_K^{(j)}\}$, which serve as the input in the E-step and convergence check step and are used as the output of the M-step. However, for practical applications, in which $d \ll N$ and $K \ll N$, their size equal $O(K * d * d)$ is much smaller than the size of array storing feature vectors equal $O(N * d)$ and the array storing posterior probabilities equal $O(N * K)$. For that reason, we focused mainly on optimization of computation performed on large arrays. The details of parallelization are as follows.

B. The E-step

In this step each OpenMP thread operates on the assigned set of feature vectors and computes the assigned set of posterior probabilities. This step is parallelized using OpenMP `omp for` directive for loop iterating over N rows of both arrays.

C. The M-step

Our method for parallelizing the M-step, similarly to [9], exploits the fact, that the M-Step involves several reduction operations. In all-to-one reduction operation [16] there are p copies of initial data item, each of them assigned to a single thread. The result of the reduction is obtained by combining the data items through an associative and commutative operation (addition in our case) and accumulating it into one data item. In other words, our implementation of M-step uses reduction to find the sum of a set of elements.

The computation of model parameters $\Theta^{(j)}$ requires two passes over the arrays storing posterior probabilities and feature vectors. Booth passes are implemented using loops parallelized with `omp for` directive. In the first pass the vectors $\mathbf{w} = [\sum_{i=1}^N h_1^{(j)}(\mathbf{x}^i), , \ldots, \sum_{i=1}^N h_K^{(j)}(\mathbf{x}^i)]$ as and $\mathbf{v}_m = \sum_{i=1}^N h_m^{(j)}(\mathbf{x}^i) * \mathbf{x}^i$, $m = 1, \ldots, K$ are obtained. The loop iterating over elements of arrays storing feature vectors and posterior probabilities is parallelized using `omp for` directive. Each OpenMP thread computes a partial sum iterating over its assigned set of data items, and at the end of the loop these sums are combined through the reduction operation. Since in OpenMP for C++ reduction operations are restricted to primitive data types (user-defined reductions will be supported in the upcoming OpenMP 4.0 standard) we have implemented our own reduction algorithm, in which the master thread collects and adds partial sums produced by all the threads. It should be noted, that this is a suboptimal method, which yields linear time complexity with respect to the number of OpenMP threads. However, the same approach is used in gcc OpenMP implementation for primitive data types [17]. After the computation of $\mathbf{w}$ and $\mathbf{v}_m$, the mixing probabilities are obtained as $\alpha_m^{(j)} = \frac{1}{N} * w_m$ and similarly mean vectors as $\mu_m^{(j)} = \frac{\mathbf{v}_m}{w_m}$.

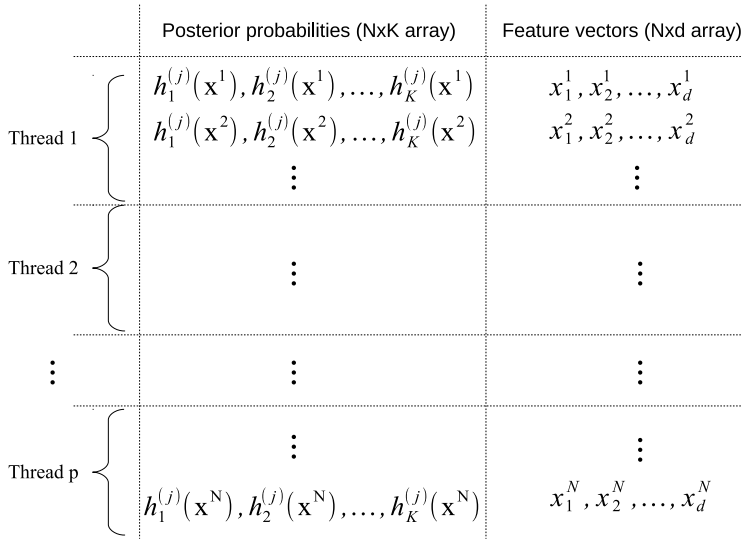

Fig. 4. Distribution of computation on large arrays storing posterior probabilities and feature vectors into OpenMP threads.

The second pass over the arrays is used to compute covariance matrices. Similarly to the first pass, the matrices $S_m = \sum_{i=1}^{N} h_m^{(j)}(\mathbf{x}^i)(\mathbf{x}^i - \mu_m^{(j)})(\mathbf{x}^i - \mu_m^{(j)})^T$ are obtained by calculating the partial sum by each OpenMP thread and adding up the partial sums by the master thread. Then, the covariance matrices are obtained as $\Sigma_m^{(j)} = \frac{S_m}{w_m}$.

D. Convergence check

In this step each OpenMP thread in a parallelized `for` loop computes a partial sum of (3) using its assigned sets of feature vectors and posterior probabilities. The partial sums are added using OpenMP `reduction` clause [11].

E. NUMA optimizations

As it was mentioned in section I, a crucial condition for achieving good performance on the NUMA system described in section IV is such placement of data and computation, that each OpenMP thread operates on data stored in a memory local (i.e, connected to the same NUMA node) to a core on which the thread is running. The Linux operating system running on NUMA machines by default allocates the memory using the *first-touch* policy [12]. This policy allocates a memory page frame locally to a core from which the first access to the page occurs. If the local memory is exhausted the system allocates a page frame from the NUMA node with a closest distance (e.g., the number of hops) from the local node. The first touch policy can be very successful, if the memory access pattern of threads does not change throughout the run of the program and the threads stay on their initial cores. Whereas the former condition is satisfied by our method of parallelization, OpenMP does not have a portable method for enforcing the latter. In our approach, we had to resort to implementation specific `GOMP_CPU_AFFINITY` environment variable [17].

To exploit the first-touch policy, at the beginning of the algorithm run we allocate the large arrays containing posterior probabilities and feature vectors using the Linux `mmap()` system call, which allows them to be stored in continuous blocks of memory. Next, both arrays are initialized in parallelized `for` loop by the team of OpenMP threads using the memory access pattern shown on Fig. 4. The initialization of other data (e.g., mixture parameters) is not critical, because their size allows them to be stored in L2/L3 cache memory. As the next section shows, this simple optimization significantly improves the scalability of our method.

VI. Experimental results

In this section the results of the experiments, in which our parallel version of the EM algorithms for GMMs was run on system described in Section IV, are presented. The parallel version was implemented using C++ language and compiled by gcc 4.7.3 compiler (which supports OpenMP 3.1) using optimization switches (-Ofast -flto -march=bdver1 -mprefer-avx128) recommended for the AMD "Interlagos" processor. The system was running an Ubuntu 12.04 server Linux distribution based on the kernel 3.2.0. The goal of the experiments was to assess the scalability of our implementation.

In the experiments we used two synthetic datasets simulated from a Gaussian mixture obtained by a generator recently proposed in [18]. We first randomly generated parameters of a single mixture consisting of $K = 20$ components and with the dimension of the feature space $d = 10$. Using this set of parameters we simulated two datasets: the dataset with 13.5 millions of feature vectors with total size 0.5 GB. and the dataset with 135 millions of feature vectors with total size of 5 GB. To conserve space we used single precision floating point format (4 bytes) for datasets (feature vectors). The posterior probabilities and mixture parameters were stored using double precision floating point format (to prevent issues with numerical precision).

To evaluate the scalability of our implementation, we run the EM algorithm, changing the number of used cores from 1

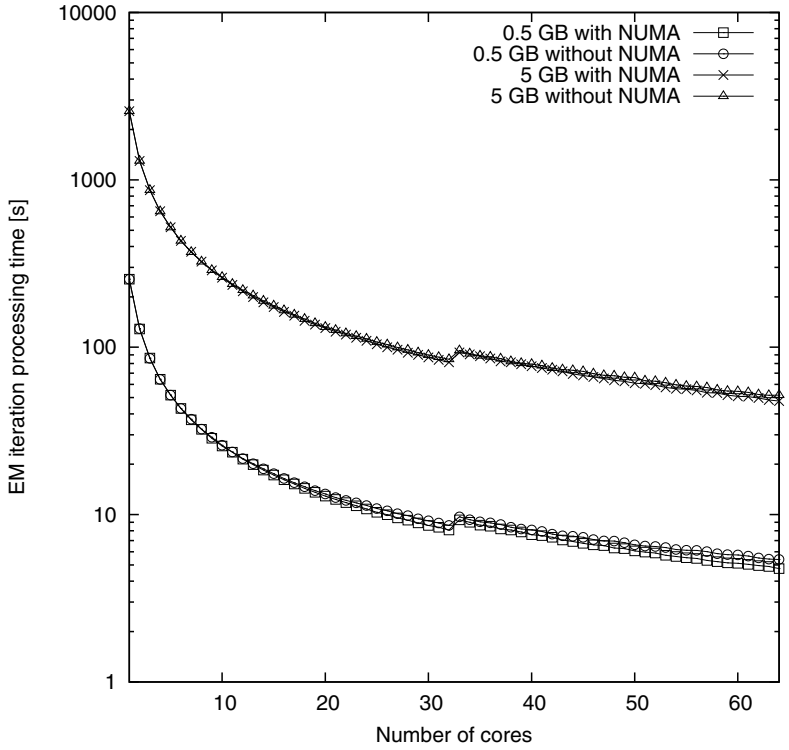

Fig. 5. Average EM iteration times for two datasets and two versions of our algorithm.

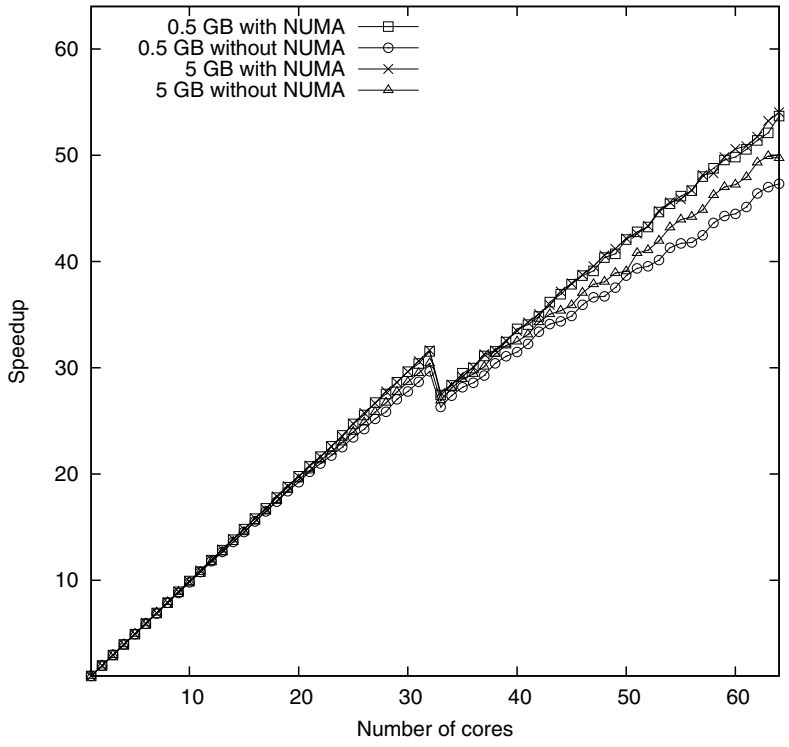

Fig. 6. Speedup curves for two datasets and two versions of our algorithm.

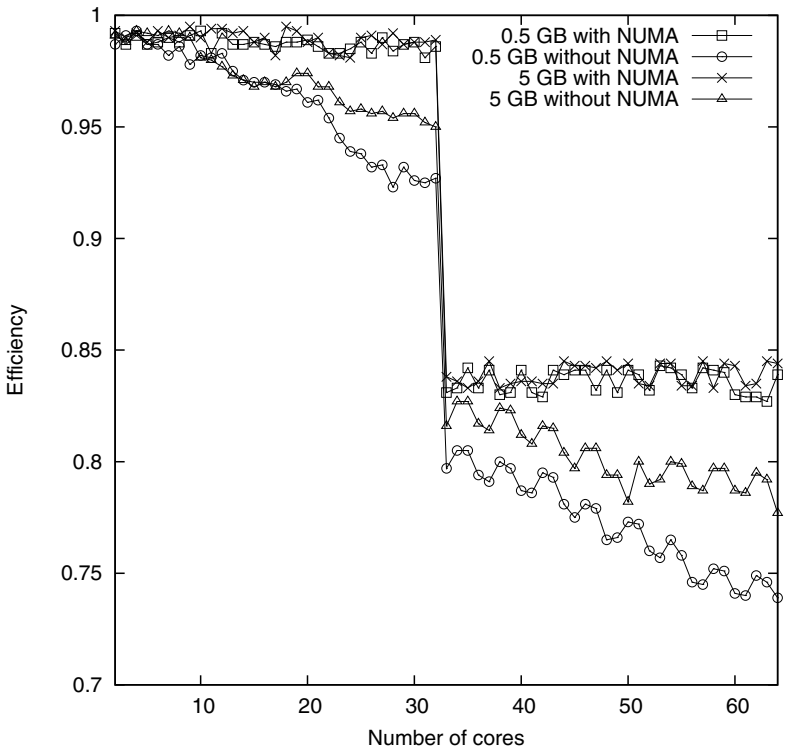

Fig. 7. Efficiency curves for two datasets and two versions of our algorithm.

to 64. In order to limit resource contention between cores in a module, when the number of cores was in range between 1 and 32 we allocated only one core from the Bulldozer module keeping the other core idle. When the number of used cores passed 32 we started to allocate the second cores from the modules. The core allocation was performed using `GOMP_CPU_AFFINITY` environment variable [17]. Using this variable, we were able to bind each thread to a single core, which assured the thread affinity, i.e., the lack of migration of OpenMP threads between the cores. The EM algorithm was run once for the given dataset and number of cores.

Fig. 5 presents the average EM iteration processing times for two datasets and two different versions of our algorithm: the version running with NUMA optimizations and the version running without them. The average EM iteration time was computed as the ratio of total algorithm runtime to the number of EM iterations needed to achieve to convergence according to (8) (for $\epsilon = 10^{-5}$ the algorithm needs 21 iterations for 0.5 GB dataset and 16 iterations for 5GB dataset). Fig. 6 shows the parallel speedup curves obtained by both versions. The speedup for p cores is defined as the ratio of runtime of serial version of the algorithm and the parallel version running on p cores. Fig. 7 presents the parallel efficiency of our approach. The efficiency is defined as the ratio of the achieved speedup to the ideal linear speedup (equal p for p cores).

The results indicate that:

- For the number of used cores in range between 1 and 32 our method with NUMA optimizations achieves near linear speedup for both datasets (efficiency is greater than 0.98).
- When we start to employ second cores from the Bulldozer modules, the efficiency drops rapidly. This phenomenon can be explained by the contention for shared resources (e.g., L2 cache and floating point units) between two cores in a module, which slows them down causing load imbalance. Nonetheless our method is able to obtain very good speedup (efficiency greater than 0.82 for both datasets).
- The NUMA optimizations significantly improve the performance of our method (the efficiency for 64 cores on non-optimized version was 0.73 for 0.5 GB dataset and 0.78 for 5 GB dataset). Moreover, for the NUMA optimized version the parallel efficiency is approximately constant (not counting the drop when the number of cores passes 32). This is not the case for the version running without NUMA optimizations. It can be seen that the efficiency decreases steadily, when the number of cores increases from 32 to 64.
- The average time of single EM iteration for 0.5 GB dataset is 4.75 seconds, whereas the average time of single EM iteration for ten times larger 5 GB dataset is 47.6 seconds. It means, that our implementation scales linearly with increased training set size.

VII. Conclusions

We have reported on parallelization of the EM algorithm designed for large multicore NUMA machines. We have experimentally shown, that our approach scales very well up to 64

cores in spite of architectural limitations of AMD Interlagos processor and the use of suboptimal reduction algorithm. We have also shown, that simple optimizations, which enable our method to effectively exploit of the *first-touch* memory allocation policy of the Linux operating system significantly improve the scalability. In future work we are going to extend our approach (using MPI) to clusters of NUMA machines and test it on an Infiniband cluster consisting of 16 64-core Dell Power Edge R815 servers, which will be built at our department.

ACKNOWLEDGMENT

This work was supported by the grant S/WI/2/2013 from Bialystok University of Technology. The author is grateful to Marek Kretowski for useful comments on an initial version of the paper.

REFERENCES

[1] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.

[2] C. Fraley and A. E. Raftery, "Model-based clustering, discriminant analysis, and density estimation," *Journal of the American Statistical Association*, vol. 97, no. 458, pp. 611–631, 2002.

[3] T. Hastie and R. Tibshirani, "Discriminant analysis by Gaussian mixtures," *Journal of the Royal Statistical Society. Series B*, vol. 58, no. 1, pp. 155–176, 1996.

[4] D. Reynolds, T. Quatieri, and R. Dunn, "Speaker verification using adapted Gaussian mixture models," *Digital Signal Processing*, vol. 10, no. 1, pp. 19–41, 2000.

[5] H. Permuter, J. Francos, and I. Jermyn, "A study of Gaussian mixture models of color and texture features for image classification and segmentation," *Pattern Recognition*, vol. 39, no. 4, pp. 695–706, 2006.

[6] R. A. Redner and H. F. Walker, "Mixture densities, maximum likelihood and the EM algorithm," *SIAM Review*, vol. 26, no. 2, pp. 195–239, 1984.

[7] P. E. López de Teruel, J. M. García, and M. E. Acacio, "The parallel EM algorithm and its applications in computer vision," in *Parallel and Distributed Processing Techniques and Applications*, 1999, pp. 571–578.

[8] R. Yang, T. Xiong, T. Chen, Z. Huang, and S. Feng, "Distrim: Parallel GMM learning on multicore cluster," in *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol. 2. IEEE, 2012, pp. 630–635.

[9] L. Glimcher and G. Agrawal, "Parallelizing EM clustering algorithm on a cluster of SMPs," in *Euro-Par 2004 Parallel Processing. LNCS 3149*. Springer, 2004, pp. 372–380.

[10] N. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for Gaussian mixture models on GPUs using CUDA," in *11th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2009, pp. 103–109.

[11] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT Press, 2007.

[12] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and thread affinity in OpenMP programs," in *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?* ACM, 2008, pp. 377–384.

[13] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York: Springer, 2006.

[14] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, pp. 6–15, 2011.

[15] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[16] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Addison-Wesley, Second Edition, 2003.

[17] The GNU OpenMP implementation. Free Software Foundation. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.7.3/libgomp/

[18] R. Maitra and V. Melnykov, "Simulating data to study performance of finite mixture modeling and clustering algorithms," *Journal of Computational and Graphical Statistics*, vol. 19, no. 2, pp. 354–376, 2010.