Computing and Informatics, Vol. 31, 2011, 247–265

PARALLEL RETRIEVAL OF DENSE VECTORS IN THE VECTOR SPACE MODEL

Tobias Berka

Department of Computer Sciences
University of Salzburg
Jakob-Haringer-Strasse 2
5020 Salzburg, Austria
e-mail: tberka@cosy.sbg.ac.at

Marian Vajteršic

Department of Computer Sciences
University of Salzburg, Austria
&
Department of Informatics, Mathematical Institute
Slovak Academy of Sciences, Bratislava, Slovakia
e-mail: marian@cosy.sbg.ac.at

Revised manuscript received 1 December 2010

Abstract. Modern information retrieval systems use distributed and parallel algorithms to meet their operational requirements, and commonly operate on sparse vectors; but dimensionality-reducing techniques produce dense and relatively short feature vectors. Motivated by this relevance of dense vectors, we have parallelized the vector space model for dense matrices and vectors. Our algorithm uses a hybrid partitioning splitting documents and features and operates on a mesh of hosts holding a block partitioned corpus matrix. We show that the theoretic speed-up is optimal. The empirical evaluation of an MPI-based implementation reveals that we obtain a super-linear speed-up on a cluster using Nehalem Xeon CPUs.

Keywords: Vector space model, symmetric multiprocessing, dense vector computations, message passing interface

Mathematics Subject Classification 2000: 15-04, 65F99, 68P20

1 INTRODUCTION

In text retrieval, we have a range of techniques for reducing the dimensionality of a corpus, such as latent semantic indexing [4] or spectral decomposition of the feature covariance matrix [8]. All of these methods approximate the original similarities amongst the sparse vectors in the original corpus with dense vectors of reduced dimensionality. Furthermore, content-based retrieval of other forms of media such as images, video or sound has to operate on signal processing features, leading to vectors that are dense and relatively low dimensional. Consequently, dense vectors play an important role in contemporary information retrieval systems.

The discrepancy between the traditional focus on sparse vector processing on one hand, and the importance of dense vectors for multimedia retrieval and dimensionality reduction techniques on the other hand, has indeed left a gap in parallel and distributed information retrieval methods. We have developed a parallel algorithm to meet this demand, which is briefly sketched as follows:

- Both features and documents are split into balanced partitions.
- The vector-matrix multiplication, which dominates the complexity of retrieval for dense vectors in the vector-space model, is performed in parallel across all feature groups.
- The results are aggregated through a parallel merge-sort across the document groups.

This paper is organized as follows: Section 2 contains a concise, mathematical definition of retrieval in the vector space model under partitioned features and documents. The resulting parallel algorithm is described in Section 3. In Section 4, we formally show that the theoretic speed-up without communication costs is linear in the number of hosts and hence optimal. We complete the performance evaluation with empiric measurements on a small cluster. Due to improved utilization of the multi-level memory hierarchy of a modern multi-core processor, we experience a so-called super-linear speed-up effect – the speed-up we obtained is actually greater than the number of processes.

2 PARALLEL APPROACH

Assume that we have a set of features $F = \{f_1, \ldots, f_m\}$ and a set of documents $D = \{d_1, \ldots, d_n\}$. Furthermore, we have a corpus matrix C, which is a column-wise assembly of the documents' vectors, formally $C = \left[\vec{d_1}, \ldots, \vec{d_n}\right] \in \mathbb{R}^{m \times n}$. Similarity between two vectors d and e is defined as the cosine of the enclosed angle

$$\mathrm{sim}(d, e) := \cos(d, e) = \frac{\langle d, e \rangle}{\|d\| \|e\|}.$$

For any two documents d_i and d_j, we can rewrite the above equation to use the corresponding components of the corpus matrix C and simplify the sums to obtain

$$\text{sim}(d_i, d_j) \quad = \quad \frac{\left(C^T C\right)_{i,j}}{\sqrt{\left(C^T C\right)_{i,i}}\sqrt{\left(C^T C\right)_{j,j}}}.$$

If we compute a *similarity matrix* $\tilde{D}$ containing the inner products of all pairs of documents using matrix multiplication, $\tilde{D} = C^T C \in \mathbb{R}^{n \times n}$, we can compute document similarity with one multiplication, one division, one square root and three matrix component load operations. When responding to a dynamic query with a vector $q \in \mathbb{R}^m$, we can compute similarity via the vector-matrix product

$$\text{sim}(q, d_i) \quad = \quad \frac{\left(qC\right)_i}{\|q\|\sqrt{\tilde{D}_{i,i}}}.$$

Now, we partition our corpus into M pairwise disjoint sets of features $F_1, \ldots, F_M$, where $\|F_i\| = m_i$, and N pairwise disjoint sets of documents $D_1, \ldots, D_N$, where $\|D_i\| = n_i$, we partition our corpus matrix into blocks $C[i,j] \in \mathbb{R}^{m_i \times n_i}$ with corresponding $\tilde{D}[i,j] \in \mathbb{R}^{m_i \times n_j}$, such that

$$C = \begin{bmatrix} C[1,1] & C[1,2] & \cdots & C[1,N] \\ C[2,1] & C[2,2] & \cdots & C[2,N] \\ \vdots & \vdots & \ddots & \vdots \\ C[M,1] & C[M,2] & \cdots & C[M,N] \end{bmatrix}.$$

We can now compute similarity between two documents $d_{u,i} \in D_u$ and $d_{v,j} \in D_v$ as

$$\begin{aligned} \text{sim}(d_{u,i}, d_{v,j}) \quad &= \quad \begin{cases} \text{sim}_1(d_{u,i}, d_{v,j}) & \ldots u = v, \\ \text{sim}_2(d_{u,i}, d_{v,j}) & \ldots u \neq v. \end{cases}, \text{ where} \\ \text{sim}_1(d_{u,i}, d_{v,j}) \quad &= \quad \frac{\sum_{l=1}^{M} \tilde{D}[l,u]_{i,j}}{\sqrt{\sum_{l=1}^{M} \tilde{D}[l,u]_{i,i} \sum_{l=1}^{M} \tilde{D}[l,u]_{j,j}}}, \\ \text{sim}_2(d_{u,i}, d_{v,j}) \quad &= \quad \frac{\sum_{l=1}^{M} \left(\vec{d}_{u,i} C[l,v]\right)_j}{\|\vec{d}_{u,i}\|\sqrt{\sum_{l=1}^{M} \tilde{D}[l,v]_{j,j}}}. \end{aligned}$$

For a previously unknown query vector $q \in \mathbb{R}^m$, split according to our partitioning into $q[1], \ldots, q[M]$, we can use the formula

$$\text{sim}(q, d_{u,i}) = \frac{\sum_{l=1}^{M} \left(q[l] C[l,u]\right)_i}{\|q[l]\|\sqrt{\sum_{l=1}^{M} \tilde{D}[l,u]_{i,i}}}.$$

Using inverted files as a sparse matrix representation for information retrieval, the feature and document partitioning approaches [6] and the hybrid partitioning strategy [11] have been successfully applied in a parallel setting [9, 10]. For dense vectors, there is a rich background in parallel matrix computations. We can compute the similarity matrices of the vector space model using parallel matrix multiplication, as described in e.g. [5] for shared memory computers, [3] for SIMD machines or [1] for bulk synchronous parallel computers. For the analysis of the similarities of an established corpus, we can use a suitable algorithm for matrix multiplication for the available hardware architecture; but we are more interested in dynamic queries and hence vector-matrix multiplication. The parallel, partitioned multiply-accumulate computation is difficult to improve on mainstream processors without specialized hardware features.

Conceptually, we think of the nodes of our cluster as being arranged as a mesh. All hosts within one row hold the same set of features and all hosts in one column the same set of documents. Every host thereby belongs to exactly one feature-group of hosts, i.e. the row, and one document-group of hosts, i.e. the column. Note that this organizational concept does not make any assumptions about the communication topology among hosts, but merely serves as a convenient metaphor for discussing our algorithm. If we compute vector similarity as discussed above, the complete query process can be sketched as follows:

1. For corpus queries, we must first locate the query document and retrieve its vector. First, we broadcast the document identifier to all nodes of the system.

   Those nodes which hold a part of the document's vector then broadcast their vector segment to all nodes within the same row.

   Once all nodes hold the applicable part of the document vector they can proceed with the local query processing. We have depicted this process in Figure 1.
2. For vector queries, we simply broadcast the entire query vector, discard the irrelevant features and compute the local dot products and square vector lengths, as shown in Figure 2.
3. We then compute the global results in three steps sketched in Figure 3:
   (a) Compute the individual sums given in the mathematical model in parallel across the feature partitions (i.e. rows of hosts).
   (b) Compute the cosine similarity from the individual sums.
   (c) Sort the results of the different document partitions into a single, global result list.

3 PARALLEL ALGORITHM

A full implementation of the algorithm contains numerous details to deal with message passing and various management aspects of an information retrieval service,

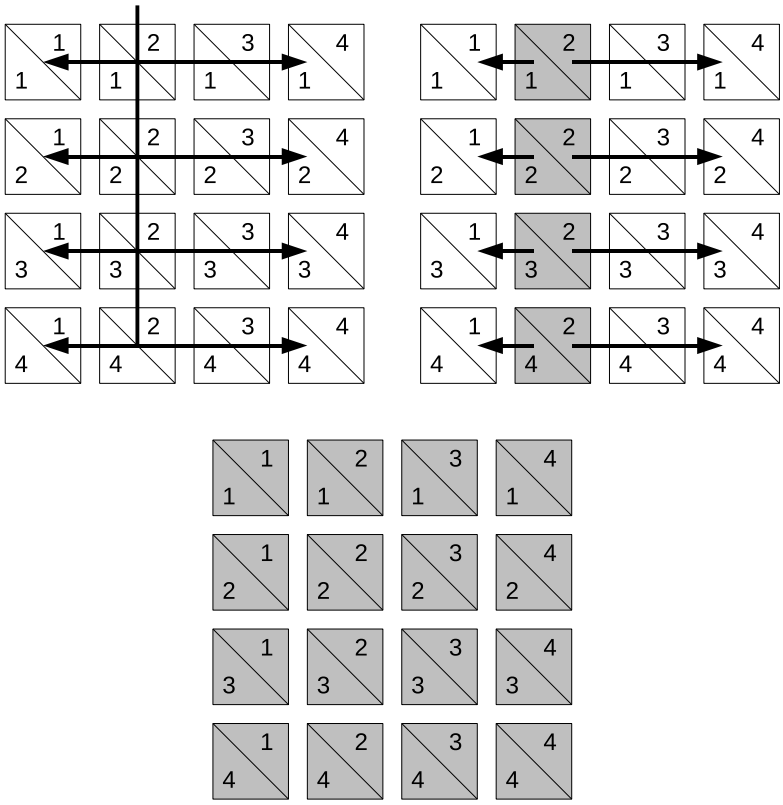


Fig. 1. Communication flow for corpus query initialization. The identifier for the query document is broadcast to all nodes (top left). The nodes within that document group, which hosts the requested document, broadcast the local part of the document vector within their respective feature groups (top right). Now all hosts can proceed with the local query processing (bottom).

which are detailed in [2]. We give a bird-eye view of the query process by examining the global behavior and the operations on the partitioned corpus matrix as pseudo-code in Algorithm 1.

We have used the Message Passing Interface (MPI) as communication middleware to implement our algorithm. It has served us especially well for two of the collective communication operations: broadcasting to all hosts and computing the parallel sum. The broadcast functionality allows us to perform a broadcast without having to deal with platform-specific details, because vendor-specific MPI implementations can be expected to provide an efficient realization. The parallel sum is provided by MPI as a collective operation on vectors and requires only a few lines of code to implement. One noteworthy drawback of MPI for our purposes was the lack of collective operations on lists. We have written a straightforward implementation

if *query with a known document* d **then**
  broadcast the document identifier to all nodes;
  if d *is locally available* **then**
    fetch the local part of the document's vector;
    broadcast the vector to all nodes on the same row;
  else
    receive the broadcast;
  end
else if *query with an unknown document's vector* $\vec{q}$ **then**
  broadcast the query vector from the root to all nodes;
  drop all features of $\vec{q}$ that are not in the local feature partition;
end
for $h = 1, \ldots, M$ *and* $k = 1, \ldots, N$ **do in parallel**
  if *a vector* $\vec{q}$ *has been received* **then**
    compute

$$R[h,k] = \begin{pmatrix} (qC[h,k])_1 & \|q[h]\|^2 & \tilde{D}[h,k]_{1,1} \\ \vdots & \vdots & \vdots \\ (qC[h,k])_i & \|q[h]\|^2 & \tilde{D}[h,k]_{i,i} \\ \vdots & \vdots & \vdots \\ (qC[h,k])_{n_k} & \|q[h]\|^2 & \tilde{D}[h,k]_{n_k,n_k} \end{pmatrix};$$

  else if *the document* $d = d_j$ *was locally available* **then**
    compute

$$R[h,k] = \begin{pmatrix} \tilde{D}[h,k]_{1,j} & \tilde{D}[h,k]_{j,j} & \tilde{D}[h,k]_{1,1} \\ \vdots & \vdots & \vdots \\ \tilde{D}[h,k]_{i,j} & \tilde{D}[h,k]_{j,j} & \tilde{D}[h,k]_{i,i} \\ \vdots & \vdots & \vdots \\ \tilde{D}[h,k]_{n_k,j} & \tilde{D}[h,k]_{j,j} & \tilde{D}[h,k]_{n_k,n_k} \end{pmatrix};$$

  end
end
for $k = 1, \ldots, N$ **do in parallel**
  compute the parallel sum $R[k] = \sum_{h=1}^{M} R[h,k]$;
  compute the list of document-similarity pairs $S[k] = (S[k]_1, \ldots, S[k]_n)$, where

$$S[k]_i = \left(d_i, \frac{R[k]_{i,1}}{\sqrt{R[k]_{i,2} R[k]_{i,3}}} \right);$$

end
merge sort the lists of document-similarity pairs $S[k]$ across all columns to generate a complete result list S;
return S;

Algorithm 1: Parallel Query Algorithm

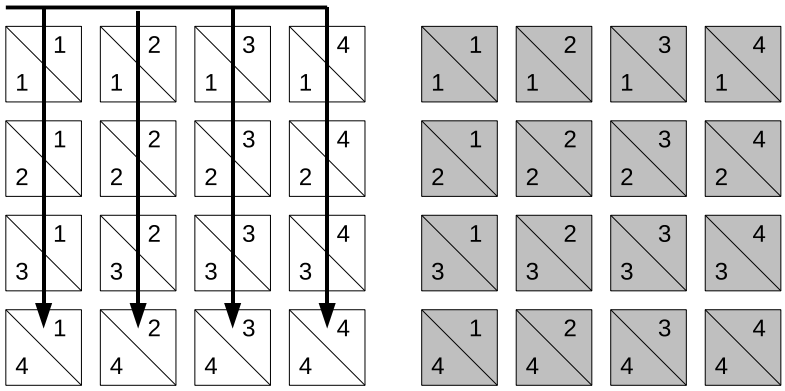


Fig. 2. Communication flow for vector query initialization. The query vector is broadcast to all nodes (left). The individual nodes then discard all unnecessary features and proceed with the local query processing (right).

of a parallel merge sort algorithm with a flat-tree communication structure using MPI's point-to-point communication primitives.

Communication in MPI is based on the concept of a communicator, which provides a common communication domain for a subset of all MPI processes in an application. Since we require global, row-wise and column-wise communication, we use the initial, global communicator to derive sub-domains for every row and column of the mesh. We call these sub-domains *row* or *column communicators* and refer to the global communicator as *mesh communicator*. That said, we can now describe the query process for symmetric multiprocessing using message-passing from the leap-frog perspective of a single node by giving those algorithms, which are executed on every individual processor. Algorithms 2 and 3 describe the query initiation process for corpus and vector queries up to and including the local query processing. Algorithm 4 describes the query result aggregation, which computes the similarity scores for the individual document partitions before sorting them into a single list of search results.

Let us now examine the performance, effectiveness and scalability of our algorithm and the prototype implementation in detail.

4 RESULTS

For our discussion of the algorithmic complexity, let us first assume an optimal, balanced distribution of features and documents for the sake of simplicity. To initiate a query, we must conduct one or two subsequent broadcasts. For a corpus query, we must first announce the document we wish to search with, before the holding process can send it to all other hosts. For vector queries, it suffices to send the query vector to all processes. Either way, these broadcasts can be realized in $O(\log M + \log N) =$

```
if row = 0 and col = 0 then
    broadcast/send query document ID on mesh communicator;
else
    broadcast/receive query document ID on mesh communicator;
end
if ID available in local document partition then
    broadcast/send query vector on row communicator;
    conduct local corpus query to produce R[row, col];
else
    broadcast/receive query vector on row communicator;
    conduct local vector query to produce R[row, col];
end
```

Algorithm 2: Initiating a Corpus Query

```
if row = 0 then
    if col = 0 then
        broadcast/send query vector on column communicator;
    else
        broadcast/receive query vector on column communicator;
    end
    drop non-local feature components;
    broadcast/send query vector on row communicator;
else
    broadcast/receive query vector on row communicator;
end
conduct local vector query to produce R[row, col];
```

Algorithm 3: Initiating a Vector Query

```
reduce/sum operation on column communicator to compute R[col];
if row = 0 then
    convert R[col] to S[col];
    run a parallel merge sort on the row communicator;
    if col = 0 then
        return S;
    end
end
```

Algorithm 4: Query Result Aggregation

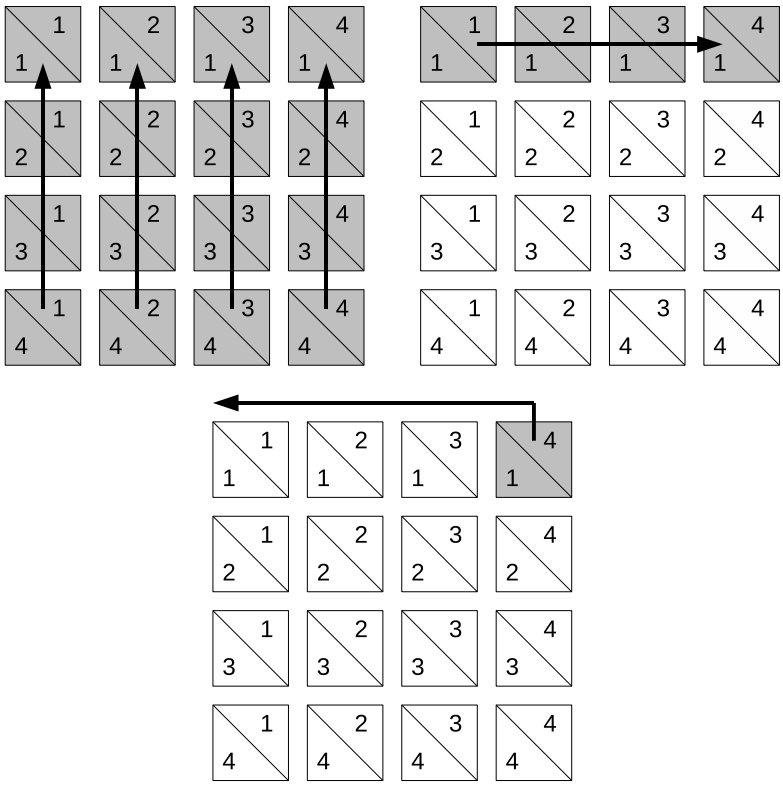


Fig. 3. Communication flow for result aggregation. The parallel matrix-vector product is computed using a parallel sum within document groups (top left). The group-specific hit lists are then sorted into a single, global hit list using a parallel merge-sort within a single feature group (top right). The final result is then returned to the inquiring client (bottom).

$O(\log MN)$ serial message sends in any modern point-to-point network. The entire query vector must be distributed, therefore the size of the message is of $O(m)$. The vector must be restricted to the locally available features, which has a time complexity of $O(m)$.

The local processing for both vector and corpus queries is bound by a time complexity of $O\left(mn/(MN)\right)$. In a serial system, this phase has a time complexity of $O(mn)$ for vector queries. For corpus queries, it depends largely on the available memory. If the system has enough memory to store $O\left(n^2\right)$ document similarity values, then we can pre-compute all intra-corpus similarity scores using matrix multiplication. After that, a corpus query can be answered in $O(n)$, because we must copy n document similarity scores. However, this approach is problematic for large values of n, because the memory required grows with n^2. It very quickly becomes

necessary to exchange the memory cost for a computational cost of $O(mn)$ and answer corpus queries dynamically using matrix-vector multiplication. For corpus queries with pre-computed similarity scores our system has a clear disadvantage, as we experience a speed-up of $O(MN/m)$, which is typically less than 1 since the number of features m can be expected to be greater than the number of hosts MN. This theoretic speed-up would be greater than 1 only for exceptionally large parallel systems or trivially low numbers of features. In both cases, the cost of communication would be out of proportion. For vector queries we obtain a theoretic speed-up of $O(MN)$, which is linear in the number of hosts and thus optimal.

Once we have computed the local inner products and lengths, we must combine these distributed results and form an aggregate, global result. This is done in two subsequent steps. First, we compute the parallel sum of the local query data. Then, we sort our results by descending similarity. Regarding the complexity, we observe the following:

1. Parallel sum: Parallel addition of the local results requires additional $O(\log M)$ message sends. Each host must transmit the local dot product and length for each of its documents, hence the message size is of $O\left(n/N\right)$. On every step of this parallel activity, we have to perform $3n/N$ additions. Hence the time complexity is $O(n/N \log M)$ for the parallel computation of the sum and $O(n/N)$ to convert the distributed results into a similarity list.
2. Merge sort: For the parallel merge sort, we initially sort the local lists. We use the asymptotic bound for a comparing sort algorithm, which sorts a list of n elements in $O(n \log n)$ steps [7]. On every individual node, we first sort a list of n/N elements, giving us a time complexity of

$$O\left(\frac{n}{N}\log\frac{n}{N}\right) = O\left(\frac{n}{N}\log n - \frac{n}{N}\log N\right).$$

   For simplicity, assume that we have $N = 2^L$ columns of hosts. In general, the bounds derived here also hold for $L = \lceil \log N \rceil$. In the first step, every second host must merge a total of $n2^{-(L-1)}$ entities – its own entities and those of another host. In the second step, every fourth host merges $n2^{-(L-2)}$ entities. In all further steps the number of hosts is halved and the number of entities is doubled. So for 2^L hosts we require

$$\sum_{k=0}^{L-1} \frac{n}{2^k} = \frac{2^L - 1}{2^{L-1}} n = 2n\frac{N-1}{N}$$

   serial steps of computation. The number of messages sent sequentially is in the order of $O(\log N)$. Consequently, the average message size is of $O(2n(N - 1)/(N \log N))$.

In a serial retrieval system, we only have to sort the global results in $O(n \log n)$ steps. If we form the speed-up by dividing this time complexity by the total time

complexity of all parallel steps and analyze the asymptotic behavior of the resulting expression, we see that our parallel result aggregation comes to a speed-up of $O(N)$ for $n \to \infty$; but this limit is reached at a logarithmic rate. So despite the fact that this speed-up is linear in N, we would not expect an almost linear effect to manifest except for substantially large document collections.

Summarizing, we note that the time complexity of the entire query process is dominated by the vector-matrix product and is of $O(mn)$. Using our parallel algorithm we obtain a speed-up of $S = O(MN)$ without communication costs, which is theoretically optimal. Let us now examine the behavior of our prototype implementation on a real system.

To evaluate the speed-up empirically, we have conducted preliminary timing measurements on a cluster consisting of eight nodes equipped with an Intel Xeon E5520 CPU clocked at 2.27 GHz with 4 cores equipped with 48 GiB of RAM. We used between 8 processes on 2 machines and 32 processes on 8 machines, using 4 threads per processor, one per physically available core. While we did not explicitly use the hyper-threading feature, which provides two virtual threads per physical core, we kept it enabled throughout the tests to allow the operating system to use these extra threads. Measurements have been performed using a test corpus containing random vectors with 1 024, 2 048 and 4 096 features. For reasons of scope, we only report our results for 1 024 and 4 096 features, because speed-up and efficiency for 2 048 features follow the general trend. The number of documents has been sampled over different ranges so that the problem size, i.e. the product of the number of features and document, is equal for all sampling points. For 1 024 features we have conducted experiments with 131 072 to 1 048 576 documents. With 4 096 features we have used a quarter thereof, 32 768 to 262 144 documents.

We then conducted corpus queries *without* pre-computed similarity matrices, because the document similarity matrix $\tilde{D}$ is too big to be kept in memory for just over one million documents. We measured the complete response time from dispatch of the query to the return of the complete result list for a single process. Then, we measured the parallel execution time of the hybrid partitioning strategy using mesh layouts of 2×4 to 2×16 processes. We experimented with a variety of field sizes, but found using two feature groups most beneficial for our problem sizes.

Before we discuss our results, we also need to point out that our implementation uses 32-bit, single-precision floating point variables and arrays, because the available memory bandwidth plays an important role in the efficiency of a parallel algorithm on a multi-core CPU. While every core has its own local level 1 cache and a local but synchronized level 2 cache, they all share a limited number of memory channels to access the DRAM. The bandwidth of these channels is an important limiting factor, because all cores have to be kept supplied with data to process. The efficiency with double precision data is worse than the values reported here.

The first component required for computing the speed-up is the serial run-time. The serial processing times for our sample problem sizes are depicted in Figure 4. This diagram also gives a good visual indication of the impact of the vector-matrix product on the execution time. The time spent on sorting has hardly any impact

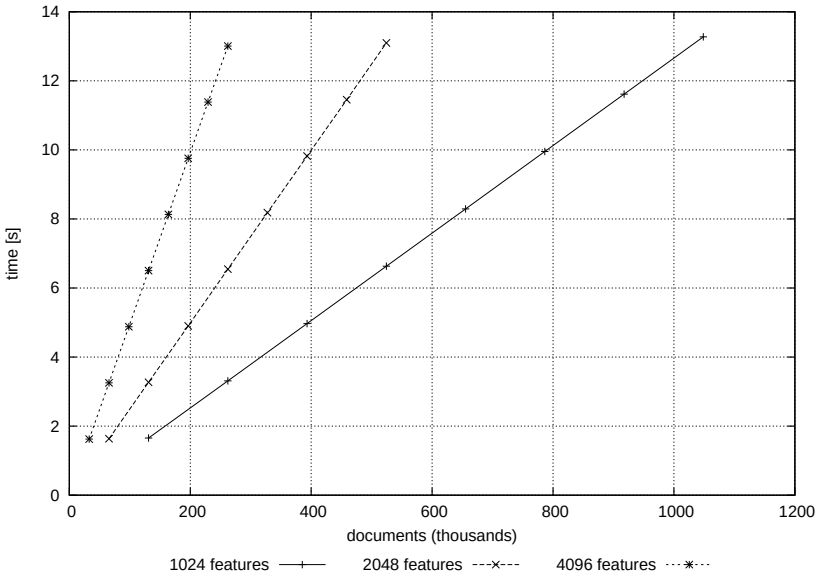


Fig. 4. Serial query processing time with 1 024, 2 048 and 4 096 features

on the execution time, despite the fact that the number of documents differed by as much as a factor of 4 for different feature sizes.

Figure 5 shows the results for the document partitioning strategy with 1 024 features. While the parallel efficiency is generally between 80 % and 100 % (see Figure 6) we fail to obtain any super-linear speed-up effects (see Figure 7).

In Figures 8 to 13 we can clearly see the super-linear speed-up effect for both 1 024 and 4 096 features. The speed-up is almost linear across the sampled range of documents until the main memory is almost full, meaning that the parallel algorithm does not strongly suffer from adverse effects as the number of documents grows. The higher speed-up and better efficiency with 4 096 features suggests that a host mesh with two feature groups can not only handle such a number of features, but is in fact more efficient as the number of features increases. This is again fortunate, as the low number of feature groups allows us to use more document host groups to accelerate the parallel merge sort and deal with large numbers of documents more effectively.

5 SUMMARY AND CONCLUSION

In this paper, we have applied a hybrid partitioning of features and documents to the mathematical formulation of the vector space model and derived a parallel algorithm based on message passing. We analyzed the theoretic speed-up without communication cost and showed that it is linear in the number of processes and

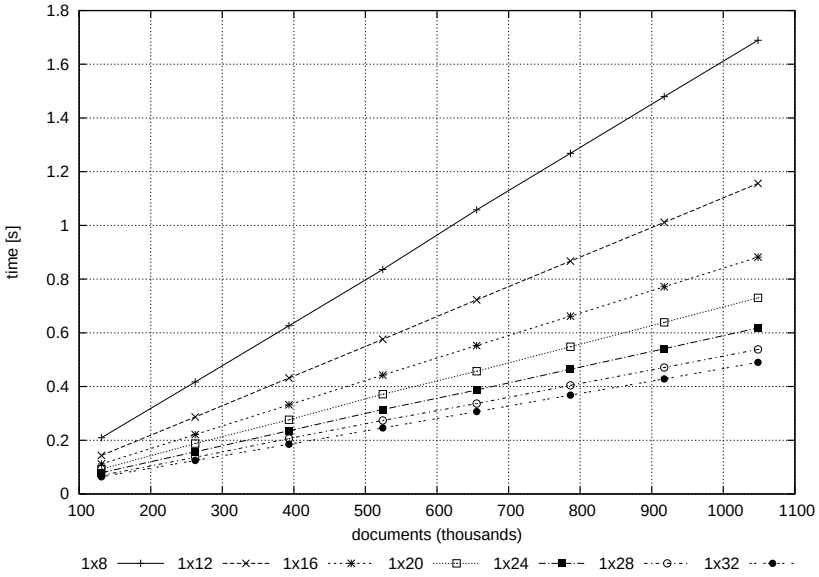


Fig. 5. Query processing time for 1 024 features using a document partitioning strategy

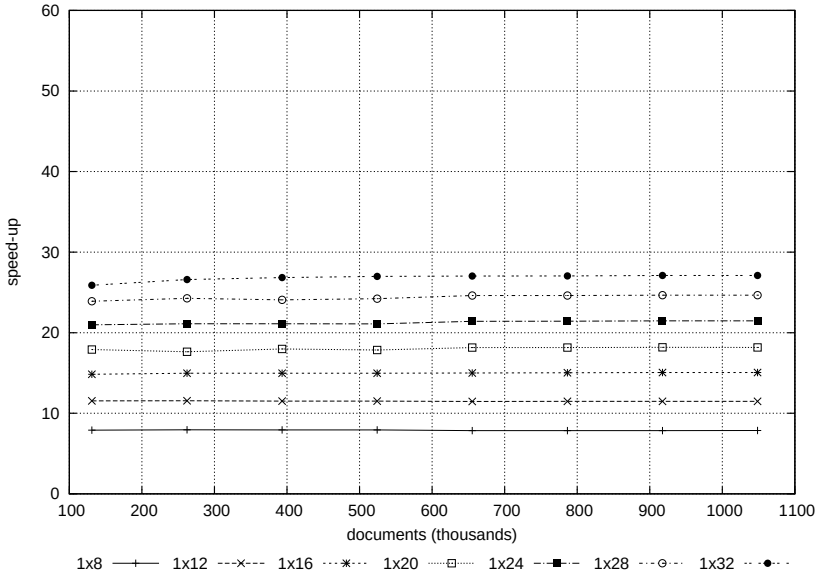


Fig. 6. Speed-up for 1 024 features using a document partitioning strategy

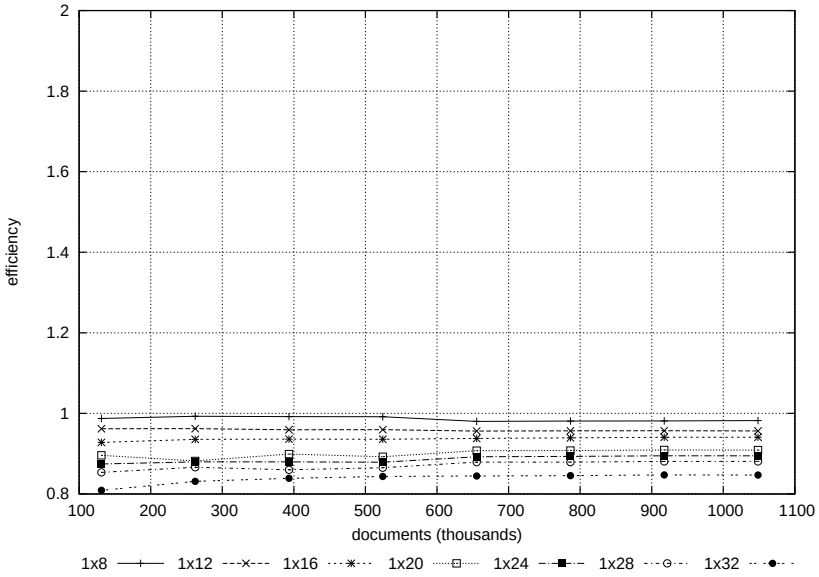


Fig. 7. Efficiency for 1 024 features using a document partitioning strategy

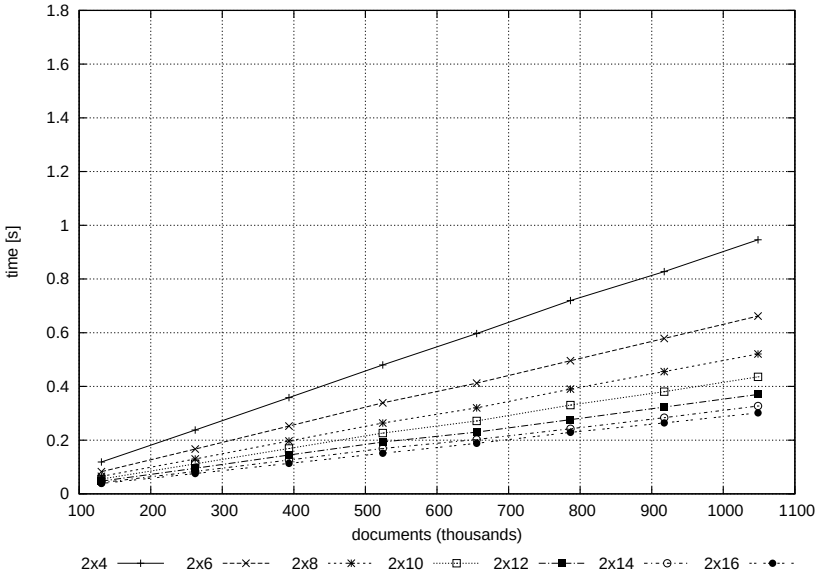


Fig. 8. Query processing time for 1 024 features using a hybrid partitioning strategy

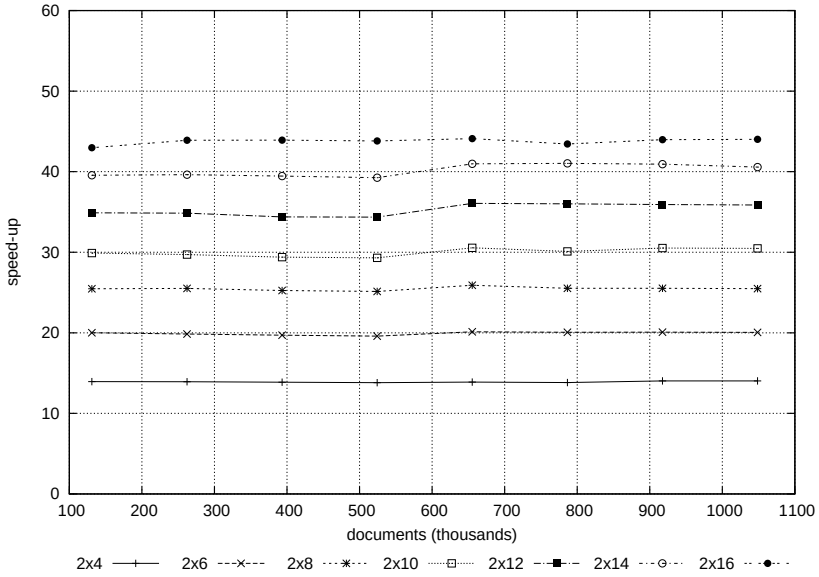


Fig. 9. Speed-up for 1 024 features using a hybrid partitioning strategy

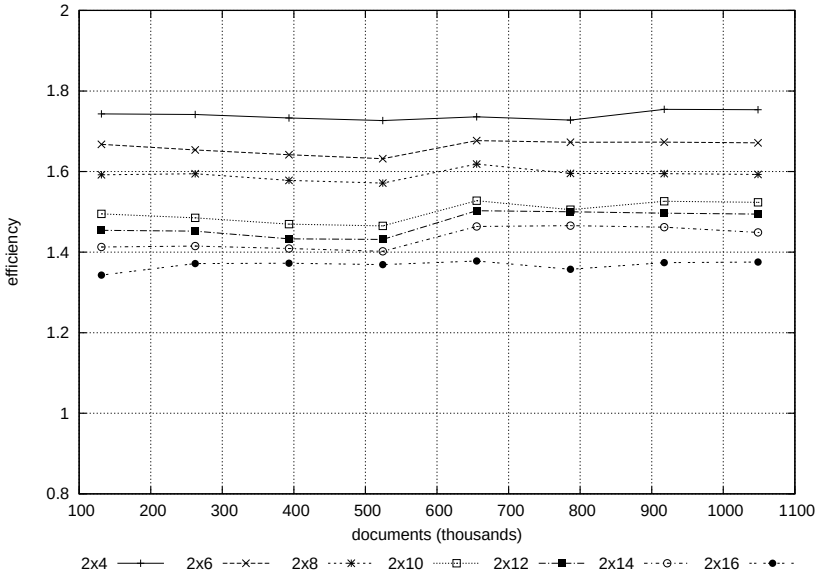


Fig. 10. Efficiency for 1 024 features using a hybrid partitioning strategy

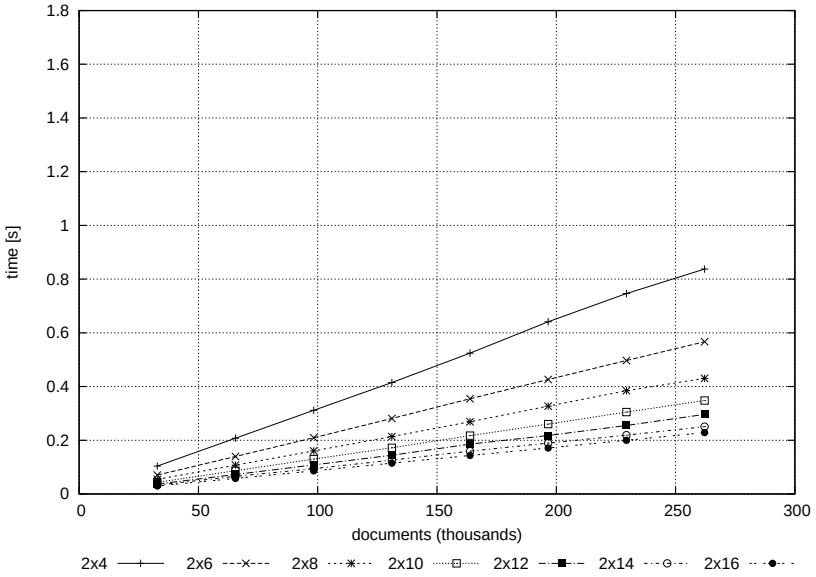


Fig. 11. Query processing time for 4 096 features using a hybrid partitioning strategy

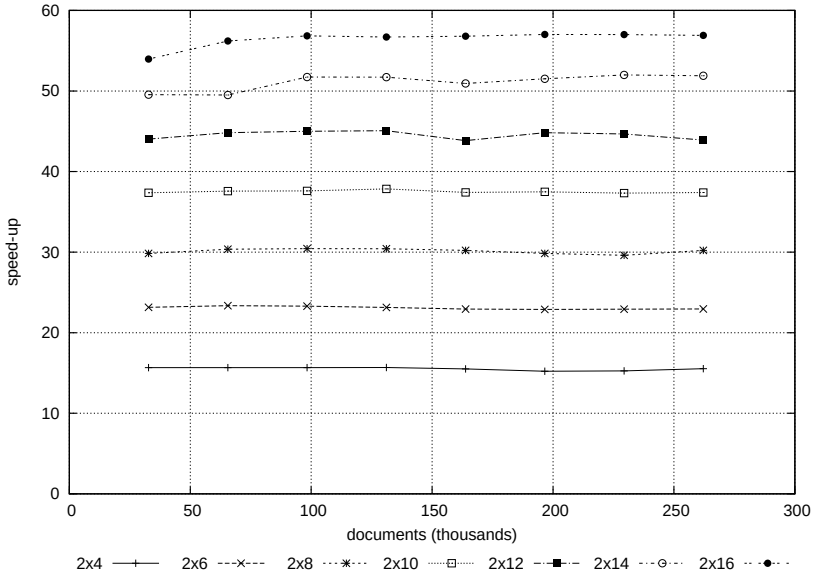


Fig. 12. Speed-up for 4 096 features using a hybrid partitioning strategy

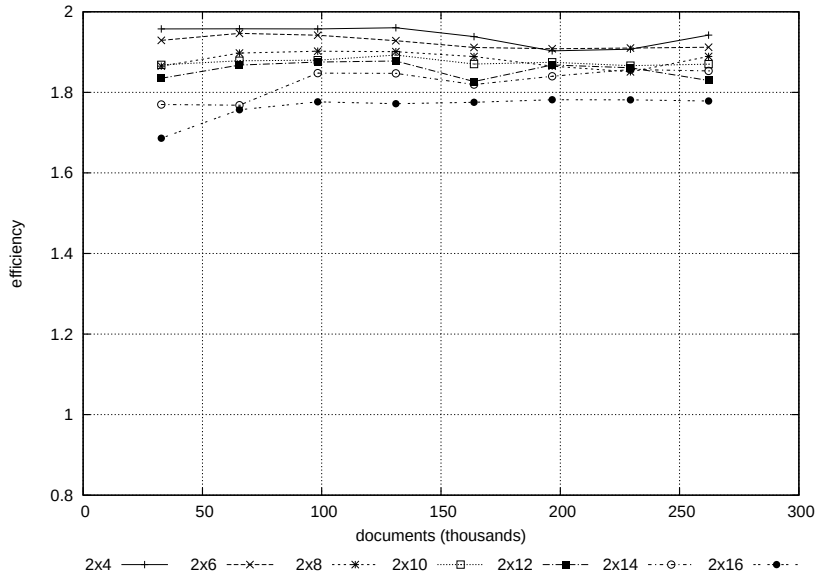


Fig. 13. Efficiency for 4 096 features using a hybrid partitioning strategy

hence optimal. Then, we empirically evaluated the performance of an MPI-based implementation and found that our algorithm delivers a super-linear speed-up due to efficient utilization of the memory hierarchy. These measurements indicate that we can confidently utilize all available cores, without having to fear the bottleneck of the shared DRAM channels or destructive interference on the level 2 data caches. Lastly, our data illustrates that we can increase the number of documents until we reach the practical limit of the computers main memory without having to fear a loss of efficiency.

REFERENCES

[1] Beran, M.: Decomposable Bulk Synchronous Parallel Computers. In: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics, pp. 349–359, 1999.

[2] Berka, T.: Distributed Image Retrieval on the Grid using the Vector Space Model. Master's Thesis, University of Salzburg, Department of Computer Sciences, Austria, 2009.

[3] Bjørstad, B.—Manne, F.—Sørevik. T.—Vajteršic, M.: Efficient Matrix Multiplication on SIMD Computers. SIAM Journal of Matrix Analysis and Applications, Vol. 13, 1992, No. 1, pp. 386–401.

[4] Deerwester, S. C.—Dumais, S. T.—Landauer, T. K.—Furnas, G. W.—Harshman, R. A.: Indexing by Latent Semantic Analysis. Journal of the Society for Information Science, Vol. 41, 1990, No. 6, pp. 391–407.

[5] Francomano, E.—Tortorici-Macaluso, A.—Vajteršic, M.: Implementation Analysis of Fast Matrix Multiplication Algorithms on Shared Memory Computers. Computers and Artificial Intelligence, Vol. 14, 1995, No. 3, pp. 299–313.

[6] Jeong, B.-S.—Omiecinski, E.: Inverted File Partitioning Schemes in Multiple Disk Systems. IEEE Transactions on Parallel and Distributed Systems, Vol. 6, 1995, No. 2, pp. 142–153.

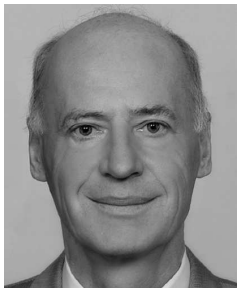
[7] Knuth, D. E.: The Art of Computer Programming. Volume 3: (2^{nd} Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[8] Kobayashi, M.—Aono, M.—Takeuchi, H.—Samukawa, H.: Matrix Computations for Information Retrieval and Major and Outlier Cluster Detection. Journal of Computational and Applied Mathematics, Vol. 149, 2002, No. 1, pp. 119–129.

[9] MacFarlane, A.—McCann, J.—Robertson, S.: PLIERS: A Parallel Information Retrieval System Using MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, Vol. 1697, 1999, pp. 674–674.

[10] Orlando, S.—Perego, R.—Silvestri, F.: Design of a Parallel and Distributed Web Search Engine. ArXiv Computer Science e-prints, 2004.

[11] Xi, W.—Sornil, O.—Luo, M.—Fox, E. A.: Hybrid Partition Inverted Files: Experimental Validation. In: Proceedings of the 6^{th} European Conference on Research and Advanced Technology for Digital Libraries, pp. 422–431, 2002.



Marian Vajteršic graduated in numerical mathematics from Comenius University, Bratislava (Slovak Republic) in 1974. He received his C. Sc. (candidate of sciences) degree in mathematics from the same university in 1984 and he defended the Dr. Sc. (doctor of sciences) degree in there 1997. In 1995, he obtained the habilitation degree in numerical mathematics and parallel processing from the University of Salzburg (Austria). His research activity is focused on the area of parallel numerical algorithms for high-performance computer systems. He is the author of two monographs, co-author of four other books and of more than 100 scientific papers. Since 1974, he is with the Slovak Academy of Sciences in Bratislava, Slovakia. As a Visiting Professor he had been with the universities of Vienna, Bologna, Milan, Linz, Salzburg, Amiens and Munich. Since 2002 he is a Full Professor at the Department of Computer Sciences at the University of Salzburg, Austria.

Tobias Berka received a B. Sc. and M. Sc. with honors in computer sciences from the University of Salzburg, Austria, in 2008 and 2009. He is now a Ph. D. candidate and student of Prof. Vajteršic. His research is in the field of parallel algorithms and parallel matrix computations with applications in information retrieval and data mining.