Large-Scale Multi-Dimensional Document Clustering on GPU Clusters

Yongpeng Zhang, Frank Mueller
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
Email:mueller@cs.ncsu.edu

Xiaohui Cui, Thomas Potok
Oak Ridge National Laboratory
Computational Sciences and Engineering Division
Oak Ridge, TN 37831
Email:cuix@ornl.gov

Abstract

Document clustering plays an important role in data mining systems. Recently, a flocking-based document clustering algorithm has been proposed to solve the problem through simulation resembling the flocking behavior of birds in nature. This method is superior to other clustering algorithms, including k-means, in the sense that the outcome is not sensitive to the initial state. One limitation of this approach is that the algorithmic complexity is inherently quadratic in the number of documents. As a result, execution time becomes a bottleneck with large number of documents.

In this paper, we assess the benefits of exploiting the computational power of Beowulf-like clusters equipped with contemporary Graphics Processing Units (GPUs) as a means to significantly reduce the runtime of flocking-based document clustering. Our framework scales up to over one million documents processed simultaneously in a sixteen-node moderate GPU cluster. Results are also compared to a four-node cluster with higher-end GPUs. On these clusters, we observe 30X-50X speedups, which demonstrate the potential of GPU clusters to efficiently solve massive data mining problems. Such speedups combined with the scalability potential and accelerator-based parallelization are unique in the domain of document-based data mining, to the best of our knowledge.

1. Introduction

Document clustering, or text clustering, is a sub-field of data clustering where a collection of documents are categorized into different subsets with respect to document similarity. Such clustering occurs without supervised information, *i.e.*, no prior knowledge of the number of resulting subsets or the size of each subset is required. Clustering analysis in general is motivated by the explosion of information accumulated in today's Internet, *i.e.*, accurate and efficient analysis of millions of documents is required within a reasonable amount of time. A recent flocking-based algorithm [4] implements the clustering process through the simulation of mixed-species birds in nature. In this algorithm, each document is represented as a point in a two-dimensional Cartesian space. Initially set at a random coordinate, one point interacts with its neighbors according to a clustering criterion, *i.e.*, typically the similarity metric between documents. This algorithm is particularly suitable for dynamical streaming data and is able to achieve global optima, much in contrast to our algorithmic solutions [16].

The inherently quadratic computational complexity in the number of documents and the large memory footprints, however, make efficient implementation of flocking for document clustering a challenging task. Yet, the parallel nature of such a model bears the promise to exploit advances in data-parallel accelerators for distributed simulation of flocking. Previous research has demonstrated more than five times speedups using a single GPU card over a single-node desktop for several thousands documents [1]. This testifies to the benefits of GPU architectures for highly parallel, distributed simulation of individual behavioral models. Nonetheless, such accelerator-based parallelization is constrained by the size of the physical memory of the accelerating hardware platform, *e.g.*, the GPU card.

In this research, our goal is to process at least one million documents at a time. This unprecedented scale imposes significant memory consumption that far exceeds the memory capacity of a single GPU. We investigate the potential to purse our goal on a cluster of computers equipped with NVIDIA CUDA-enabled GPUs. We are able to cluster one million documents over sixteen NVIDIA GeForce GTX 280 cards with 1GB on-board memory each. Our implementation demonstrates its capability for weak scaling, *i.e.*, execution time remains constant as the amount of documents is increased at the same rate as GPUs are added to the processing cluster. We have also developed a functionally equivalent multi-threaded MPI application in C++ for performance comparison. The GPU cluster implementation shows dramatic speedups over the C++ implementation, ranging from 30X to more than 50X speedups.

Related research to our work can be divided into two categories: (1) fast simulation of group behavior and (2) GPU-accelerated implementations of document clustering. (1) The first basic flocking model was devised by Reynolds [13]. Here, each individual is referred as a "boid". Three rules are quantified to aid the simulation of flocks: separation, alignment and cohesion. Since document clustering groups documents in different subsets, a multiple-species flocking (MSF) model is developed by Cui *et al.* [4]. This model adds a similarity check to apply only the separation rule to non-similar boids. A similar algorithm is found by Momen *et al.* [8] with many parameter tuning options. Computation time becomes a concern as the need to simulate large numbers of individuals prevails. Zhou *et al.* [19] describe a way to parallelize the simulation of group behavior. The simulation

space is dynamically partitioned into P divisions, where P is the number of available computing nodes. A mapping of the flocking behavioral model onto streaming-based GPUs is presented by Erra *et al.* [5] with the objective of obstacle avoidance. This study predates the most recent language/runtime support for general-purpose GPU programming, such as CUDA, which allows simulations at much larger scale.

(2) Recently, data-parallel co-processors have been utilized to accelerate many computing problems, including some in the domain of massive data clustering. One successful acceleration platform is that of Graphic Processing Units (GPUs). Parallel *data* mining on a GPU was assessed early on by Che *et al.* [2], Fang *et al.* [7] and Wu *et al.* [17]. These approaches rely on k-means to cluster a large space of data points. Since the size of a single point is small (*e.g.*, a constant-sized vector of floating point numbers to represent criteria such as similarity in our case), memory requirements are linear to the size of individuals (data points), which is constrained by the local memory of a single GPU in practice. In document clustering, the size of each document varies and can reach up to several kilo-bytes. Therefore, document clustering imposes an even higher pressure on memory usage. Unfortunately, many accelerators, including GPUs, do not share memory with their host systems, nor do they provide virtual memory addressing. Hence, there is no means to automatically transfer data between GPU memory and host main memory. Instead, such memory transfers have to be invoked explicitly. The overhead of these memory transfers, even when supported by DMA, can nullify the performance benefits of execution on accelerators. Hence, a thorough design to assure well-balanced computation on accelerators and communication / memory transfer to and from the host computer is required, *i.e.*, overlap of data movement and computation is imperative for effective accelerator utilization.

The contributions of this work are three-fold:

- We apply multiple-species flocking (MSF) simulation in the context of large-scale document clustering on *GPU clusters*. We show that the high I/O and computational throughput in such a cluster meets the demanding computational and I/O requirements.
- In contrast to previous work that targeted GPU clusters [6], [3], our work is one of the first to utilize GPU clusters to accelerate *massive data mining applications*, to the best of our knowledge.
- The solid speedups observed in our experiments are reported *over the entire application* (and not just by comparing kernels without considering data transfer overhead to/from accelerator). They clearly demonstrate the potential for this application domain to benefit from acceleration by GPU clusters.

The rest of the paper is organized as follows. We begin with the background description in Section 2. The programming model design and the detailed implementation are presented in Section 3. In Section 5, we show various speedups of GPU clusters against CPU clusters in different configurations. The work is summarized in Section 6.

2. Background Description

In this section, we describe the algorithmic steps of document clustering, namely similarity preprocessing and cluster detection, and discuss details of the target programming environments.

2.1. Similarity Preprocessing

The first step in document clustering, similarity preprocessing, is based on data obtained from a large corpus of text articles. The MSF model relies on a global similarity metric between any pair of documents. This involves the following preprocessing steps:

- *Document tokenization*: This step consists of stripping out unused tags, stop words, numbers and punctuations. The purpose of this step is to remove noise in the similarity calculation.
- *Word stemming*: We apply Porter's algorithm [10], which is the *de factor* standard for stemming. This is part of a term normalization process for English-language documents that removes common morphological and inflectional endings from words. It also increases the accuracy of the final result.
- *TF-ICF (term frequency, inverse corpus frequency) calculation*: In contrast to the standard TF-IDF [15] calculation used in assessing document similarity, TF-ICF does not require term frequency information from other documents within the processed document collections. Instead, it pre-builds the ICF table by sampling a large amount of existing literature off-line. Selection of corpus documents for this training set is critical as similarities between documents of a later test set are only reliable if both training and test sets share a common base dictionary of terms (words) with a similar frequency distribution of terms over documents. Once the ICF table is constructed, ICF values can be looked up very efficiently for each term in documents while TF-IDF would require dynamic calculation of these values. The TF-ICF approach enables us thus to generate document vectors in linear time [11].

Having converted each document into a document vector that holds the values of each unique term's normalized TF-ICF value, we apply the cosine similarity metric to calculate the similarity between any pair of documents i and j:

$$Sim_{i,j} = \sum_{k} |TFICF_{k,i} - TFICF_{k,j}|^2 \qquad (1)$$

for k over all terms of both document i and j.

Though the above preprocessing steps are integral parts of the flocking-based document clustering algorithm, their execution time is negligible compared to the flocking simulation step. Thus, for the rest of the paper, we will focus on the design and analysis of flocking simulation only.

2.2. Flocking-based document clustering

The second step in document clustering is to form groups of individuals that share certain criteria. In flocking-based clustering, the behavior of a boid (individual) is based only on its neighbor flock mates within a certain range. Reynolds [12] describes this behavior in a set of three rules. Let $\vec{p_j}$ and $\vec{v_j}$ be the position and velocity of boid j. Given a boid noted as x, suppose we have determined N of its neighbors within radius r. The description and calculation of the force by each rule is summarized as follows:

- *Separation*: steer to avoid crowding local flock mates

$$\vec{f_{sep}} = -\sum_{i}^{N} \frac{\vec{p_x} - \vec{p_i}}{r_{i,x}^2} \tag{2}$$

  where $r_{i,x}$ is the distance between two boids i and x.
- *Alignment*: steer towards the average heading of local flock mates

$$\vec{f_{ali}} = \frac{\sum_i^N \vec{v_i}}{N} - \vec{v_x} \tag{3}$$

- *Cohesion*: steer to move toward the average position of local flock mates

$$\vec{f_{coh}} = \frac{\sum_i^N \vec{p_i}}{N} - \vec{p_x} \tag{4}$$

The three forces are combined to change the current velocity of the boid. In case of document clustering, we map each document as a boid that participates in flocking formation. For similar neighbor documents, all three forces are combined. For non-similar neighbor documents, only the *separation* force is applied.

2.3. GPU and CUDA

In our study, the target computing environment for flocking-based simulation is a cluster of accelerators, or more specifically GPUs in a cluster. Historically, GPU development has mainly been driven by increasing demands for faster and more realistic graphics effects. Since graphics is a niche, albeit a very influential one, that drives the progress in GPU architectures, much attention has been paid to fast and independent vertex rendering. The computational rendering engines of GPUs can generally be utilized for other problem domains as well, but their effectiveness depends much on the suitability of numerical algorithms within the target domain for GPUs.

In recent years, GPUs have attracted more and more developers who strive to combine high performance, lower cost and reduced power consumption as an inexpensive means for solving complex problems. This trend is expedited by the emergence of increasingly user-friendly programming models, such as NVIDIA's CUDA, AMD's Stream SDK and OpenCL. Our focus lies on the former of these models.

CUDA is a C-like language that allows programmer to execute programs on NVIDIA GPUs by utilizing their streaming processors. The core difference between CUDA programming and general-purpose programming is the capability and necessity to spawn massive number of threads. Threads are grouped into *warps* as basic thread scheduling units [9]. The same code is executed by threads in the same *warp* on a given streaming processor. As these GPUs do not provide caches, memory latencies are hidden through several techniques: (a) Each streaming processor contains a small but fast on-chip shared memory that is exposed to programmers. (b) Large register files enable instant hardware context switch between *warps*. This facilitates the overlapping of data manipulation and memory access. (c) Off-chip global memory accesses issued simultaneously by multi-threads can be accelerated by coalesced memory access, which requires aligned access pattern for consecutive threads in *warps*.

In this work, we describe the design and evaluation of flocking-based clustering for CUDA-programmed GPU devices distributed over a cluster of host compute nodes. Our approach exploits the massive throughput offered by GPUs as the major source of speedup over clusters of conventional desktops.

2.4. MPI

The document flocking algorithm is not an embarrassingly parallel algorithm as it requires exchange of data between nodes. We utilize MPI as a means to exchange data between nodes. MPI is the dominant programming model in the high-performance computation domain. It provides message passing utilities with a transparent interface to communicate between distributed processes without considering the underlying network configurations. It is also the *de factor* industrial standard for message passing that offers maximal portability. In this work, we incorporate MPI as the basic means to communicate data between distributed computation nodes. We also combine MPI communication with data transfers between host memory and GPU memory to provide a unified distributed object interface that will be discussed later.

3. Design and Implementation

3.1. Programming Model for Data-parallel Clusters

We have developed a programming model targeted at message passing for CUDA-enabled nodes. The environment is motivated by two problems that surface when explicitly programming with MPI and CUDA abstraction in combination:

- Hierarchical memory allocation and management have to be performed manually, which often burdens programmers.
- Sharing one GPU card among multiple CPU threads can improve the GPU utilization rate. However, explicit multi-threaded programming not only complicates the code, but may also result in inflexible designs, increased complexity and potentially more programming pitfalls in terms of correctness and efficiency.

To address these problems, we have devised a programming model that abstracts from CPU/GPU co-processing and mitigates the burden of the programmer to explicitly program data movement across nodes, host memories and device memories. We next provide a brief summary of the key contributions of our programming model (see [18] for a more detailed assessment):

- We have designed a *distributed object interface* to unify CUDA memory management and explicit message passing routines. The interface enforces programmers to view the application from a data-centric perspective instead of a task-centric view. To fully exploit the performance potential of GPUs, the underlying run-time system can detect data sharing within the same GPU. Therefore, the network pressure can be reduced.
- Our model provides the means to spawn a flexible number of host threads for parallelization that may *exceed* the number of GPUs in the system. Multiple host threads can be automatically assigned to the same MPI process. They subsequently share one GPU device, which may result in higher utilization rate than single-threaded host control of a GPU. In applications where CPUs and GPUs co-process a task and a CPU cannot continuously feed enough work to a GPU, this sharing mechanism utilizes GPU resources more efficiently.
- An interface for advanced users to control thread scheduling in clusters is provided. This interface is motivated by the fact that the mapping of multiple threads to physical nodes affects performance depending on the application's communication patterns. Predefined communication patterns can simply be selected so that communication endpoints are automatically generated. More complex patterns can be supported through reusable plug-ins as an extensible means for communication.

We designed and implemented the flocking-based document clustering algorithm in GPU clusters based on this GPU cluster programming model. In the following, we discuss several application-specific issues that arise in our design and implementation.

3.2. Flocking Space Partition

The core of the flocking simulation is the task of neighborhood detection. A sequential implementation of the detection algorithm has $O(N^2)$ complexity due to pair-wise checking of N documents. This simplistic design can be improved through space filtering, which prunes the search space for pairs of points whose distances exceed a threshold.

One way to split the work into different computational resource is to assign a fixed number of documents to each available node. Suppose there are N documents and P nodes. In every iteration of the neighborhood detection algorithm, the positions of local documents are broadcast to all other nodes. Such partitioning results in a lower communication overhead proportional to the number of nodes, and the detection complexity is reduced linearly by P per node for a resulting overhead of $O(N^2/P)$.

Instead of partitioning the documents in this manner, we break the virtual simulation space into row-wise slices. Each node handles just those documents located in the current slice. Broadcast messages that are previously required are replaced by point-to-point messages in this case. This partitioning is illustrated in Figure 1. After document positions are updated in each iteration, additional steps are performed to divide all documents into three categories. *Migrating documents* are those that have moved to a neighbor slice. *Neighbor documents* are those that are on the margin of the current slice. In other words, they are within the range of the radius r of neighbor slices. All other are *internal documents* in the sense that they do not have any effects on the documents in other nodes. Since the velocity of documents is capped by a maximal value, it is impossible for the migrating documents to cross an entire slice in one timestep. Both the migrating documents and neighbor documents are transferred to neighbor slices at the beginning of the next iteration. Since the neighborhood radius r is much smaller than the virtual space's dimension, the number of migrating documents and neighbor documents are expected to be much smaller than that of the internal documents.

Sliced space partitioning not only splits the work nearly evenly among computing nodes but also reduces the algorithmic complexity in sequential programs. Neighborhood checks across different nodes are only required for neighbor documents within the boundaries, not for internal documents. Therefore, on average, the detection complexity on each node reduces to $O(N^2/P^2)$ for slides partitioning, which is superior to traditional partitioning with $O(N^2/P)$.

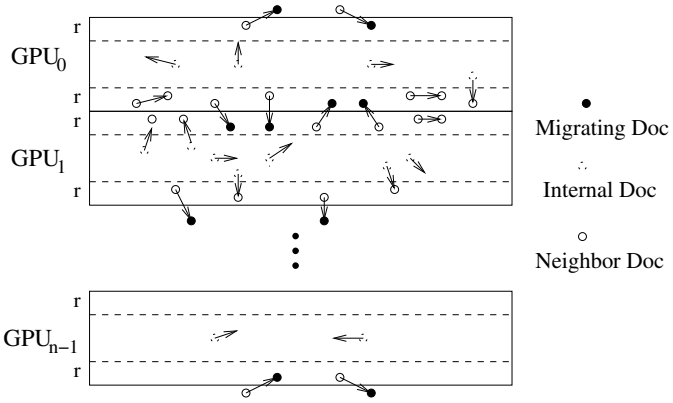


Fig. 1. Simulation Space Partition

3.3. Document Vectors

An additional benefit of MSF simulation is the similarity calculation between two neighbor documents. Similarities could be pre-calculated between all pairs and stored in a triangular matrix. However, this is infeasible for very large N because of a space complexity of $O(N^2/2)$, which dauntingly

exceeds the address space of any node as N approaches a million. Furthermore, devising an efficient partition scheme to store the matrix among nodes is difficult due to the randomness of similarity look-ups between any pair of nearby documents. Therefore, we devote one kernel function to calculating similarities in each iteration. This results in some duplicated computations, but this method tends to minimize the memory pressure per node.

The data required to calculate similarities is a document vector consisting of an index of each unique word in the TF-ICF table and its associated TF-ICF values. To compute the similarity between two documents, as shown in Equation (1), we need a fast method to determine if a document contains a word given the word's TF-ICF index. Moreover, the fact that we need to move the document vector between neighbor nodes also requires that the size of the vector should be kept small.

The approach we take is to store document vectors in an array sorted by the index of each unique word in the TF-ICF table. This data structure combines the minimal memory usage with a fast parallel searching algorithm. Riech [14] describes an efficient algorithm to calculate the cosine similarities between any two sorted arrays. But this algorithm is iterative in nature and not suitable for parallel processing.

We develop an efficient CUDA kernel to calculate the similarity of two documents given their sorted document vectors as shown in Algorithm 1. The parallel granularity is set so that each block takes one pair of documents. Document vectors are split evenly by threads in the block. For each assigned TF-ICF value, each thread determines if the other document vector contains the entry with the same index. Since the vectors are sorted, a binary search is conducted to lower the algorithmic complexity logarithmic time. A reduction is performed at the end to accumulate differences.

Algo 1: Document Vector Similarity (CUDA Kernel)

```
// calculate the similarities between two DocVecs
__device__ void docVecSimilarity(DocVec* lhs,
                                 DocVec *rhs, float *output) {
  float sim(0.0f);
  float commonSim(0.0f);
  for (int i = 0; i < lhs→NumEntries; i += blockIdx.x) {
      float tficf = biSearch(entry, rhs→vectors);
      sum += pow(entry→tficf − tficf, 2);
      commonSim += pow(tficf, 2);
  }
  // ... reduce to threadIdx.x(0), store in sum
  __syncthreads();
  if (threadIdx.x == 0) {
      sum −= commonSim;
      sum = sqrtf(sum);
      // write to global memory
      *output = sum;
  }
}

__device__ float biSearch(VecEntry *entry,
                          DocVector *vector) {
  int idx = entry→index;
  int leftIndex = 0;
  int rightIndex = vector→NumEntries;
  int midIndex = vector→NumEntries/2;
  while(true) {
      int docIdx;
      docIdx = vector→vectors[midIndex].index;
      if (docIdx < idx)
        leftIndex = midIndex + 1;
      else if (docIdx > idx)
        rightIndex = midIndex − 1;
      else
        break;

      if (leftIndex > rightIndex)
        return 0.0f;
      midIndex = (leftIndex + rightIndex)/2;
  }
  return vector→vectors[midIndex].tficf;
}
```

3.4. Message Data Structure

In sliced space partitioning, each slice is responsible to generate two sets of messages for the slices above and below. The corresponding message data structures are illustrated in Figure 2. The document array contains a header that enumerates the number of neighbors and migrating documents in the current slice. Their global indexes, positions and velocities are stored in the following array for neighborhood detection in a different slice. Due to the various sizes of each document's TF-ICF vector and the necessity to minimize the message size, we concatenate all vectors in a vector array without any padding. The offset of each vector array is stored in a metadata offset array for fast access. This design offers efficient parallel access to each document's information.

3.5. Optimizations

The algorithmic complexity of sliced partitioning decreases quadratically with the number of partitions (see Section 3.2). For a system with a fixed number of nodes, a reduction in complexity could be achieved by exploiting multi-threading within each node. However, in practice, overhead increases as the number of partitions become larger. This is particularly this case for communication overhead. As we will see in Section 5, the effectiveness of such performance improvements differs from one system to another.

At the beginning of each iteration, each thread issues two non-blocking messages to its neighbors to obtain the neighboring and migrating documents' statuses (positions) and their vectors. This is followed by a neighbor detection

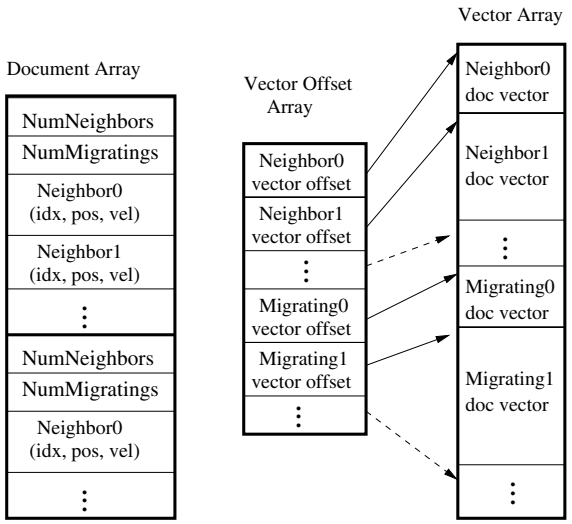


Fig. 2. Message Data Structures

function that searches its neighbor documents within a certain range for each internal document and migrated document. The search space includes every internal, neighbor and migrating document. We can split this function into three sub-functions: (a) internal-to-internal document detection; (b) internal-to-neighbor/migrating document detection and (c) migrating-to-all document detection. Sub-function (a) does not require information from other nodes. We can issue this kernel in parallel with communication. Since the number of internal documents is much larger than neighbor and migrated documents, we expect the execution time for sub-function (a) to be much larger than that of (b) or (c). From the system's point of view, either the communication or neighbor detection functions affects the overall performance.

One of the problems in simulating massive documents via the flocking-based algorithm is that as the virtual space size increases, the probability of flock formation diminishes as similar groups are less likely to meet each. In nature-inspired flocking, no explicit effort is made within simulations to combine similar species into a unique group. However, in document clustering, we need to make sure each cluster has formed only *one group* in the virtual space in the end without flock intersection. We found that an increase in the number of iterations helps in achieving this objective. We also dynamically reduce the size of the virtual space throughout the simulation. This increases the likelihood of similar groups to merge when they become neighbors.

3.6. Work Flow

The work flow for each space partition at an iteration is shown in Figure 3. Each thread starts by issuing asynchronous messages to fetch information from neighboring threads. Messages include data such as positions of the documents that have migrated to the current thread and documents at the margin of the neighbor slices. Those documents' TF-ICF vectors are encapsulated in the message for similarity calculation purposes, as discussed later.

Internal-to-internal document detection can be performed in parallel with message passing (see Section 3.5). The other two detection routines, in contrast, are serialized with respect to message exchanges. Once all neighborhoods are detected, we calculate the similarities between the documents belonging to the current thread and their detected neighbors. These similarity metrics are utilized to update the document positions in the next step where the flocking rules are applied.

Once the positions of all documents have been updated, some documents may have moved out the boundary of the current partition. These documents are removed from the current document array and form the messages for neighboring threads for the next iteration. Similarly, migrated documents received through messages from neighbors are appended to the current document array. This post-processing is performed in the last three steps in Figure 3.

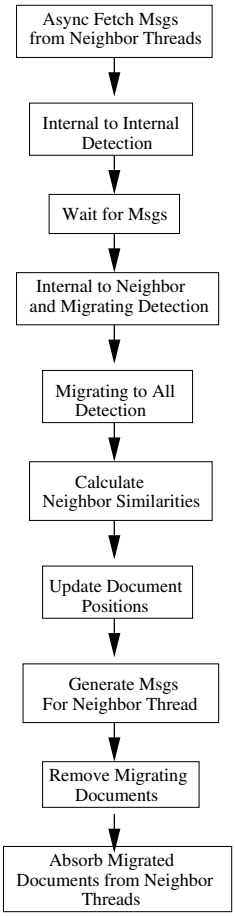
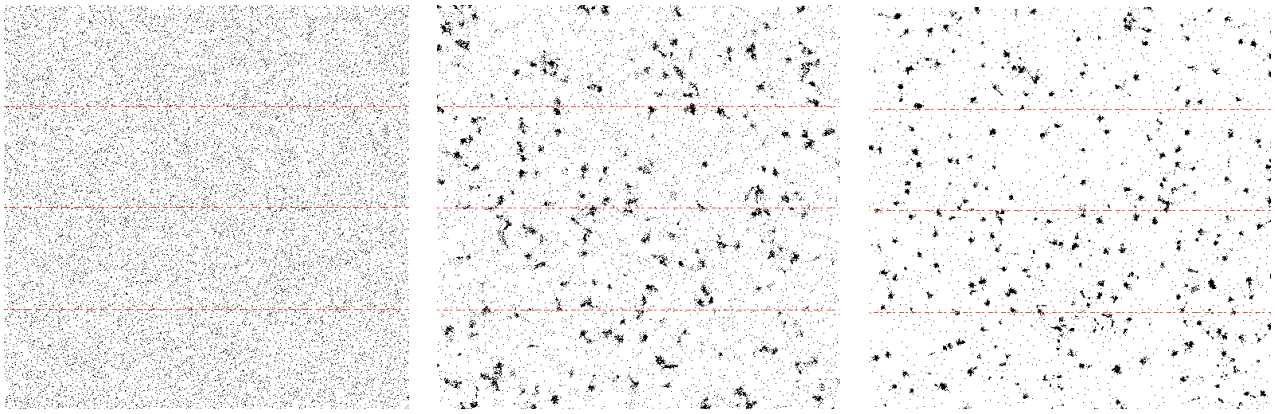


Fig. 3. Work Flow for a Thread in Each Iteration

4. Experimental Framework

To assess the effectiveness of our advanced document clustering approach, we compare executions on a GPU-accelerated



(a) Initial State (b) At Iteration 50 (c) At Iteration 500

Fig. 4. Clustering 20K Documents in 4 GPUs

	large GPU Clusters(NCSU)	large CPU clusters(NCSU)	small GPU clusters(ORNL)	small CPU clusters(ORNL)
Nodes	16	16	4	4
CPU	AMD Athlon Dual Core	AMD Athlon Dual Core	Intel Quad Q6700	Intel Quad Q6700
CPU Frequency	2.0 GHz	2.0 GHz	2.67 GHz	2.67 GHz
System Memory	SDRAM 1 GB	SDRAM 1 GB	DDR2 SDRAM 4 GB	DDR SDRAM 4 GB
GPU	16 GTX 280s	Disabled	3 Tesla C1060	Disabled
GPU Memory	1 GB	N/A	4 GB	N/A
Network	1 Gbps	1 Gbps	1 Gbps	1 Gbps

TABLE 1. Experiment Platforms

cluster with those on a functionally equivalent CPU cluster. Input documents originate from Internet news articles. The average number of unique word in each article is about 400 words. In the CPU cluster version, internal document vectors are stored in STL hash containers instead of sorted document vectors, as in GPU cluster version. This combines benefits of fast serial similarity checking with ease of programming. The message structure is the same in both implementations. Hence, functions are provided to convert STL hashes to vector arrays and vice versa.

Both implementations incorporate the same MPI library (MPICH 1.2.7p1 release) for message passing and the C++ boost library (1.38.0 release) for multi-threading in a single MPI process. The GPU version uses the CUDA 2.1 release.

5. Experimental Results

5.1. Flocking Behavior Visualization

We have implemented support to visualize the flocking behavior of our algorithm off-line once the positions of documents are saved after an iteration. The evolution of flocks can be seen in the three snapshots of the virtual plane in Figure 4, which shows a total of 20,000 documents clustered on four GPUs. Initially, documents are assigned at random coordinates in the virtual plane. After only 50 iterations, we observe an initial aggregation tendency. We also observe that the number of non-attached documents tends to decrease as the number of iterations increases. In our experiments, we observe that 500 iterations suffice to reach a stable state even for as many as a million documents. Therefore, we use 500 iterations throughout the rest of our experiments.

As Figure 4 shows, the final number of clusters in this example is quite large. This is because our input documents from the Internet cover widely divergent news topics. The resulting number is also a factor of the similarity threshold used throughout the simulation. The smaller the threshold is / the more strict the similarity check is, the more groups we will be formed through flocking.

5.2. Performance

We first compare the performance of individual kernels on an NVIDIA GTX 280 GPU hosted on a AMD Athlon 2 GHz Dual Core PC. We focus on two of the most time-consuming kernels: detecting neighbor documents (detection for short) and neighbor document similarity calculation (similarity for short). Only the GPU kernel is measured in this step. The execution time is averaged over 10 independent runs. Each run measures the first clustering step (first iteration in terms of Figure 4) to determine the speedup over the CPU version starting from the initial state. The speedup at different document sizes is shown in Figure 5. We can see that the similarity kernel on the GPU is about 45 times faster than on a CPU at almost all document sizes. For the detection kernel, the GPU is fully utilized once the document size exceeds 20,000, which

gives a raw speedup of over 300X.

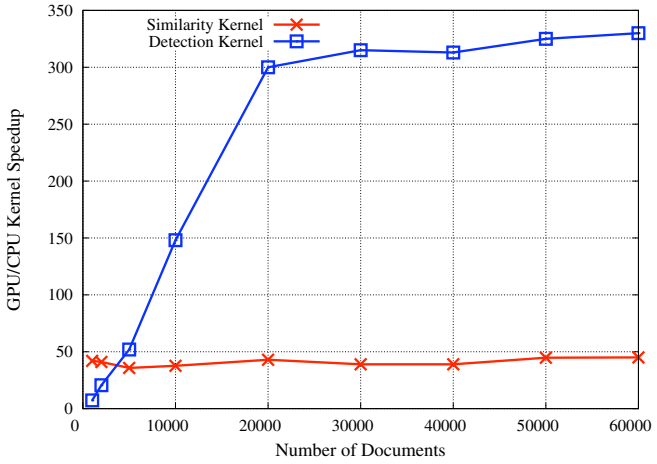


Fig. 5. Speedups for Similarity and Detection Kernels

We next conducted experiments on two clusters located at NCSU and ORNL. On both clusters, we conducted test with and without GPUs enabled (see hardware configurations in Table 1). The NCSU cluster consists of sixteen nodes with CPUs and GPUs of lower RAM capacity for both CPU and GPU, while the ORNL cluster consists of fewer nodes with larger RAM capacity. As mentioned in Section 3.1, our programming model supports a flexible number of CPU threads that may exceed the number of GPUs on our platform. Thus, multiple CPU threads may share one GPU. In our experiments, we assessed the performance for both one and two CPU threads per GPU. Figure 6 depicts the results for wall-clock time on the NCSU cluster. The curve is averaged over the execution for both one and two CPU threads per GPU. The error bar shows the actual execution time: the maximum/minimum represent one/two CPU threads per GPU, respectively. With increasing of number of nodes, execution time decreases and the maximal number of documents that can be processed at a time increases. With 16 GTX 280s, we are able to cluster one million documents within twelve minutes. The relative speedup of the GPU cluster over the CPU cluster ranges from 30X to 50X. As mentioned in Section 3.5, changing the number of threads sharing one GPU may cause a number of conflicts in resource. The benefit of multi-threading in this cluster is only moderate with only up to a 10% performance gain.

Though the ORNL cluster contains fewer nodes, its single-GPU memory size is four times larger than that of the NCSU GPUs. This enables us to cluster one million documents with only three high-end GPUs. The execution time is shown in Figure 7. The performance improvement resulting for two CPU threads per GPU is more obvious in this case: at one million documents, three nodes with two CPU threads per GPU run 20% faster than the equivalent with just one CPU thread per GPU. This follows the intuition that faster CPUs can feed more work via DMA to GPUs.

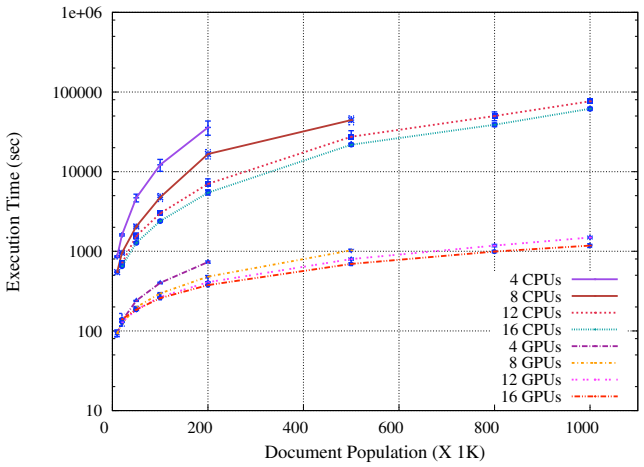


Fig. 6. GTX 280 GPUs

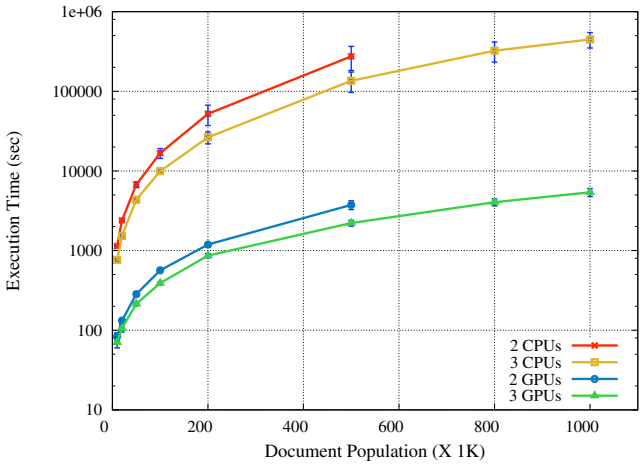


Fig. 7. Tesla C1060 GPUs

Speedups on the GPU cluster for different number of nodes and documents are shown in the 3D surface graph Figure 8 for the NCSU cluster. At small document scale (up to 200k documents), 4 GPUs achieve the best speedup (over 40X). Due to the memory constraints in these GPUs, only 200k documents can be clustered on 4 GPUs. Therefore, speedups at 500k documents are not available for 4 GPUs. For 8 GPUs, clustering with 500k documents shows an increased performance. This surface graph illustrates the overall trends: For fewer nodes (and GPUs), speedups increase rapidly over for smaller number of documents. As the number of documents increases, speedups are initially on a plane with a lower gradient before increasing rapidly, *e.g.*, between 200k and 500k documents for 16 nodes (GPUs).

We next study the effect of utilizing point-to-point messages for our simulation algorithm. Because messages are exchanged in parallel with the neighborhood detection kernel for internal documents, the effect of communication is determined by the ratio between message passing time and kernel execution

Docs(k)	5	10	20	50	100	200	500	800	1000
4 nodes	74%/9%	67%/8%	64%/5%	58%/3%	52%/1.5%	49%/0.9%	NA	NA	NA
8 nodes	67%/12%	71%/11%	65%/8%	68%/6%	62%/3.5%	56%/2%	52%/1.2%	NA	NA
12 nodes	67%/17%	69%/12%	68%/10%	71%/8%	68%/6%	63%/3%	57%/1.4%	54%/1.2%	NA
16 nodes	63%/18%	63%/13%	71%/12%	69%/9%	65%/7%	66%/4.2%	59%/1.9%	60%/1.5%	55%/1.1%

TABLE 2. Communication Percentages in GPU and CPU clusters (GPU/CPU)

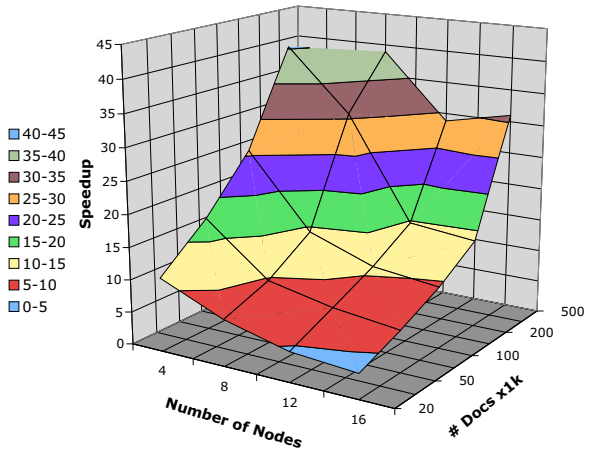


Fig. 8. Speedups on NCSU cluster

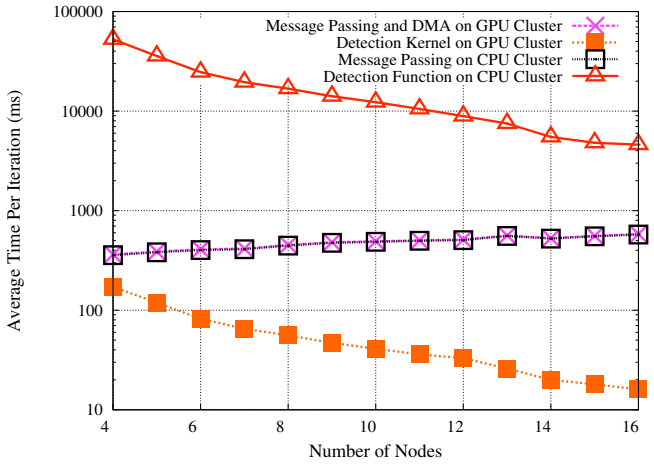


Fig. 9. Communication and Computation in Parallel

time: If the former is less than the latter, then communication is completely hidden (overlapped) by computation. In an experiment, we set the number of documents to 200k and vary the number of nodes from 4 to 16. We assess the execution time per iteration by averaging the communication time and kernel time among all nodes. The result is shown in Figure 9. For the GPU cluster, kernel execution time is always less than the message passing time. For the CPU cluster, the opposite is the case. Notice that the communication time for the GPU cluster in this graph includes the DMA duration for data transfers between GPU memory and host memory. The DMA time is almost two orders of magnitude less than that of message passing. Thus, the GPU communication/DMA curve almost coincides with that of CPU cluster's communication time, even though the latter only covers pure network time as no host/device DMA is required. This implies that internal PCI-E memory bus is not a bottleneck for GPU clusters in our experiments, which is important for performance tuning efforts. The causes for this finding are: (a) Network bandwidth is much lower than PCI-E memory bus bandwidth; and (b) messages are exchanges at roughly the same time on every node at each iteration, which may cause network congestion.

We further aggregate the time spent on message passing and divide the overall sum by the total execution time to yield the percentage of time spent on communication. For CPUs, the communication time consists of only the message passing time over the network. For GPUs, the communication time also includes the time to DMA messages to/from GPU global memory over the PCI-E memory bus. Table 2 shows the results for both GPU and CPU clusters. Generally speaking, in both cases, the ratio of communication to computation decreases as the number of documents per thread increases. The raw kernel speedup provided by GPU has dramatically increased the communication percentage. This analysis, indicating communication as a new key component for GPU clusters while CPUs are dominated by computation, implies disjoint optimization paths: faster network interconnects would significantly benefit GPU clusters while optimizing kernels even further would more significantly benefit CPU clusters.

6. Conclusion

In this paper, we present an implementation of a flocking-based document clustering algorithm accelerated by GPU clusters. Our experiments show that GPU clusters outperform CPU clusters by a factor of 30X to 50X, reducing the execution time of massive document clustering from half a day to around ten minutes. Our results show that performance gains stem from three factors: (1) acceleration through GPU calculations, (2) parallelization over multiple nodes with GPUs in a cluster and (3) a well thought-out data-centric design that promotes data parallelism. Such speedups combined with the scalability potential and accelerator-based parallelization are unique in the domain of document-based data mining, to the best of our knowledge.

7. Acknowledgement

This work was supported in part by NSF grant CCF-0429653, CCR-0237570 and a subcontract from ORNL. The

research at ORNL was partially funded by Lockheed Shared Vision research funds and Oak Ridge National Laboratory Seed Money funds.

It was partly prepared by Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, Tennessee 37831-6285, managed by UT-Battelle, LLC, for the U.S. Department of Energy under contract DE-AC05-00OR22725.

References

[1] Jesse St. Charles, Thomas E. Potok, Robert M. Patton, and Xiaohui Cui. Flocking-based document clustering on the graphics processing unit. *NICSO*, pages 27–37, 2007.

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[3] Francisco Chinchilla, Todd Gamblin, Morten Sommervoll, and Jan F Prins. Parallel n-body simulation using GPUs. Technical report, University of North Carolina at Chapel Hill, 2004.

[4] Xiaohui Cui, Jinzhu Gao, and Thomas E. Potok. A flocking based algorithm for document clustering analysis. *J. Syst. Archit.*, 52(8):505–515, 2006.

[5] Ugo Erra, Rosario De Chiara, Vittorio Scarano, and Maurizio Tatafiore. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In *Proceedings of Vision, Modeling and Visualization 2004 (VMV)*, November 2004.

[6] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Wenbin Fang, Ka K. Lau, Mian Lu, Xiangye Xiao, Chi K. Lam, Philip Y. Yang, Bingsheng He, Qiong Luo, Pedro V. Sander, and Ke Yang. Parallel data mining on graphics processors. Technical report, The Hong Kong University of Science and Technology, October 2008.

[8] S. Momen, B.P. Amavasai, and N.H. Siddique. Mixed species flocking for heterogeneous robotic swarms. In *EUROCON, 2007. The International Conference on "Computer as a Tool"*, pages 2329–2336, Sept. 2007.

[9] NVIDIA. *NVIDIA CUDA Programming Guide(Version 2.0)*, 2008.

[10] M. F. Porter. An algorithm for suffix stripping. In *Readings in information retrieval*, pages 313–316, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[11] Joel W. Reed, Yu Jiao, Thomas E. Potok, Brian A. Klump, Mark T. Elmore, and Ali R. Hurson. TF-ICF: A new term weighting scheme for clustering dynamic data streams. In *ICMLA '06: Proceedings of the 5th International Conference on Machine Learning and Applications*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Craig Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference 1999*, 1999.

[13] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[14] Konrad Rieck and Pavel Laskov. Linear-time computation of similarity measures for sequential data. *J. Mach. Learn. Res.*, 9:23–48, 2008.

[15] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. Technical report, Cornell University, Ithaca, NY, USA, 1987.

[16] Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques, 2000.

[17] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering billions of data points using GPUs. In *UCHPC-MAW '09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6, New York, NY, USA, 2009. ACM.

[18] Yongpeng Zhang, Frank Mueller, Xiaohui Cui, and Thomas Potok. A programming model for massive data parallelism with data dependencies. In *Workshop on Programming Models for Emerging Architectures*, Sep 2009.

[19] Bo Zhou and Suiping Zhou. Parallel simulation of group behaviors. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 364–370. Winter Simulation Conference, 2004.