MGMR: Multi-GPU Based MapReduce

Yi Chen[1], Zhi Qiao[1], Hai Jiang[1], Kuan-Ching Li[2], and Won Woo Ro[3]

[1] Dept. of Computer Science, Arkansas State University, USA
{yi.chen,zhi.qiao}@smail.astate.edu, hjiang@astate.edu
[2] Dept. of Computer Science & Information Engr., Providence University, Taiwan
kuancli@pu.edu.tw
[3] School of Electrical and Electronic Engineering, Yonsei University, Korea
wro@yonsei.ac.kr

Abstract. MapReduce is a programming model introduced by Google for large-scale data processing. Several studies have implemented MapReduce model on Graphic Processing Unit (GPU). However, most of them are based on the single GPU and bounded by GPU memory with inefficient atomic operations. This paper intends to develop a standalone MapReduce system, called MGMR, to utilize multiple GPUs, handle large-scale data processing beyond GPU memory limit, and eliminate serial atomic operations. Experimental results have demonstrated MGMR's effectiveness in handling large data set.

Keywords: MapReduce, multi-GPU, atomic-free, CUDA, GPUDirect.

1 Introduction

With the stagnation of CPU's performance as well as the better programmability and performance of GPU (Graphics Processing Unit), various applications have been accelerated by GPU-related programming paradigms such as CUDA 1, OpenCL 2 and Brook+ 3. The underlying reason is that GPUs' throughput - oriented computing design closely matches the characteristics of large-scale data parallel applications.

MapReduce 4 is proposed by Google to pursue simple and flexible parallel programming paradigm. With MapReduce, users only need to write Map and Reduce functions, respectively, in order to solve problems in a parallel computing manner. The low level programming details such as the ways to handle communication among data nodes are transparent to users. Data affinity across network and fault tolerance among multiple nodes can be achieved automatically.

With the success in handling large-scale data-parallel problems, many MapReduce frameworks have been implemented on parallel platforms such as multi-core CPU systems, Cell processors and GPUs 5. However, most existing GPU-based MapReduce systems put their efforts on a single GPU in a node, neglecting the multi-GPU platforms supported by advanced techniques such as GPUDirect 6. Moreover, many such systems tend to use atomic operations in GPU global memory to handle the concurrent writes among multiple threads 910. However, such atomic operations can cause serialized access of GPU memory and decrease overall performance dramatically.

J.J. Park et al. (Eds.): GPC 2013, LNCS 7861, pp. 433–442, 2013.

This paper proposes a multi-GPU MapReduce implementation, called MGMR, and makes the following contributions:

- Multiple GPUs are utilized to speed up MapReduce operation. Load balancing is achieved by distributing computations based on the capacity of all GPUs.
- Big data issue is addressed through CPU memory that normally is bigger and more extensible than GPU memory. The aggregate GPU memory is not the bottleneck anymore.
- Serial atomic operations are replaced by a parallel alternative, parallel prefix sum operation, for maximum performance gains.

The experimental results of real world applications have demonstrated that MGMR achieve significant performance gains in handling big data inputs.

The remainder of this paper is organized as follows: Section 2 briefly introduces the GPU architecture background and MapReduce framework. In Section 3, the detailed MGMR system design issues are explained for the reasons of being able to achieve the scalability. In Section 4, two real applications will be used to compare performance among CPU, single-GPU, and double-GPU MapReduce versions. Section 5 lists some related MapReduce implementations. Finally, the conclusion and future work are given in Section 6.

2 GPU and MapReduce

MGMR is developed in CUDA and based on Nvidia Fermi architecture.

2.1 Multi-GPU Architecture

Each Nvidia GPU consists of multiple streaming-multiprocessors (SMs) and can execute thousands of light-weighted hardware threads concurrently. CUDA helps map thread hierarchy onto GPU cores. Up to 512 threads are group into thread blocks that are assigned to SMs to schedule work in groups of 32 parallel threads, called warps. Extremely fast context switch with warps can help tolerate memory access latency. Each thread is assigned some registers, whereas each warp has one program counter. All threads within the same blocks can access the common shared memory. This helps synchronize threads in the same blocks, facilitate extensive reuse of on-chip data, and greatly reduce off-chip traffic.

For a machine with multiple GPUs, Nvidia GPUDirect is the technique to handle inter-GPU communication within a single system. With GPUDirect, network adapters and storage devices can directly read and write data in GPU device memory, eliminating unnecessary copies in system memory (on CPU side) to achieve significant performance improvement in data transfer. High-speed DMA engines enable this inter-GPU communication within same systems.

In Nvidia Fermi GPUs, asynchronous memory copy is another advanced feature that enables bidirectional memory copy to double data transfer bandwidth. It also helps achieve the overlapping of computation and communication 13, i.e., when GPU is busy with some calculations, DMA engines can move data around at the same time.

2.2 MapReduce Programming Model

MapReduce is widely used in various domains such as machine learning, data mining, and bioinformatics. The design goals of MapReduce include programmability, robustness and scalability. Divide-and-conquer is the basic strategy. MapReduce consists of three primary stages (*Map*, *Shuffle* and *Reduce*) where the first two can be divided further into sub-stages and the third one is indivisible. User jobs are broken apart for *Map* stage to execute. Then, *Shuffle* reorders and distributed intermediate data. Finally, *Reduce* stage merges the partial results for the final ones.

MapReduce framework tries to stay at high level and hides low-level details such as parallelism, communication, fault tolerance, and load balancing. End users only need to specify two functions: *Map* and *Reduce* for their corresponding stages. Their definitions can be abstracted as follows:

$$\text{Map: } (k_1, v_1) \rightarrow list\ (k_2, v_2) \tag{1}$$

$$\text{Reduce: (k2, list (v2))} \rightarrow \text{list (k3, v3)} \tag{2}$$

The input of *Map* is a set of key/value pairs, and the output is a list of intermediate key/value pairs. All values associated with the same key are passed to *Reduce* function that processes them for final results.

3 MGMR System Design

The target platform of MGMR is Nvidia Fermi GPU. MGMR is developed in CUDA and C++ with flexible templates. It is designed to be extensible and customizable while maintaining high occupancy of multiple GPUs. Load balancing across multiple GPUs is achieved at runtime according to hardware performance and job sizes.

3.1 Multi-GPU Utilization

All stages and sub-stages can be specified by users, and their corresponding jobs are accomplished by workers which are computers in the original MapReduce design. In MGMR, these workers will be hardware threads across multiple GPUs for load balancing.

The overview of MGMR workflow is shown in Fig. 1. The input of *Map* stage is partitioned into sets of key-value pairs, and they are assigned to workers in different GPUs simultaneously. Then, the intermediate data generated from *Map* stage are shuffled among workers across GPUs without going through CPU memory. The *Shuffle* stage incurs all-to-all communication among workers. For workers within one GPU, the communication is accomplished through commonly shared GPU global memory. For workers from different GPUs, GPUDirect enables remote GPU memory access without going through CPU memory. Performance gain is achieved there. Finally, all outputs of *Reduce* stage on multiple GPUs are copied back to CPU memory.

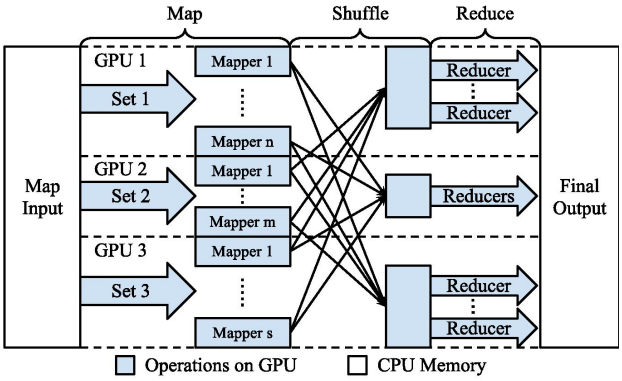

Fig. 1. MGMR workflow on multiple GPUs in single-round mode

3.2 Multi-round *Map* and *Reduce* for Big Data

Through iterative GPU activations, MGMR can handle large data set that exceeds the sum of multiple GPU memory unit sizes. However, Fig.1 only demonstrates the situation of one single round where data sets can be loaded into current GPUs. In MGMR, a data pool is allocated on CPU side in page-locked memory manner. Self-scheduling strategy is used to assign data sets onto GPUs for processing. If data cannot be processed all together by GPUs, the data pool will be used as the buffer for intermediate data in such multi-round mode. However, the single-round modes only use GPU global memory for intermediate data.

In *Map* stage, the input key-value pairs are partitioned into various sets with different sizes in the data pool. When the CPU program detects an idle GPU, it will activate one *Map* function and assign one data set over. For multi-round mode, once *Map* workers finish the work, the intermediate results will be sent back to the data pool and another group of data sets will be loaded for processing until the *Map* work is done.

In *Shuffle* stage, input/output data are placed in GPU global memory for one-round mode and in the data pool for multi-round mode.

In *Reduce* stage, *Reduce* workers will get data from GPU global memory or data pool first, and then work in self-scheduling manner. Different from Map stage, each reducer is indivisible. The input data size could be fixed or various.

3.3 *Map* Stage

The *Map* stage consists of several sub-stages: output size estimation, key-value processing and partial folder.

Output Size Estimation. MGMR estimates output size in advance to avoid memory overflow in GPU. Unlike CPU, GPU cannot dynamically allocate memory inside kernel functions. Thus, CPU program has to pre-allocate output buffer before

initiating kernel execution. To reach a balance point between higher performance and shorter programs, MGMR requires users to pre-define the structure of the output values in *struct* format. Users can do the same to the output keys as well. If they do so, a comparison function for this *struct* data type should be provided so that MGMR can sort such data items correctly.

Key-Value Processing. In this sub-stage, *Map* workers fetch input data from the data pool in self-scheduling manner and execute the user-defined *Map* function. Asynchronous memory copy such as *cudaMemcpyAsync()* is used to overlap communication and computation, i.e., both GPUs and PCIe bus will be busy at the same time.

In multi-round mode, the output data sets are needed to copy back to the data pool since GPU global memory is not big enough to contain all the intermediate results. Then, MGMR takes full advantage of bidirectional memory copying in Fermi architecture since two DMA engines work in the opposite directions. Therefore, data transfer bandwidth is doubled. Communication operations are overlapped as well.

Partial Folder. If user activates this sub-stage, the intermediate output data will be folded to reduce its size. This feature gives user a way to balance between data transmission and computation overheads. I/O bound applications can use this sub-stage to reduce data transfer cost for performance gain.

3.4 Shuffle Stage

In single-round mode, if the intermediate data (output of *Map* Stage) is small enough to put in one GPU's global memory, these key-value pairs will be sorted by radix sort provided by Nvidia Thrust library 14 in *Shuffle* stage. But if they are distributed across multiple GPUs, Parallel Sorting by Regular Sampling (PSRS) 15, also called Sample Sort, is applied to incur all-to-all broadcast through GPUDirect technique. Data will be redistributed among GPUs.

If data is too big for aggregate GPU memory and multi-round mode has to be used, the input and output data of *Shuffle* will be placed in data pool (CPU side). A CPU partition schedule will use PSRS to reorder key-value pairs and build indices in the data pool. However, GPUDirect is not necessary since the data exchange does not happen among GPUs.

The MGMR version of PSRS is implemented in four steps as follows:

1. **Local Sorting.** Each GPU is assigned a contiguous set of p items out of total n key-value pairs that will be sorted locally by radix sort.
2. **Pivot Selection and Local Data Partitioning.** On each GPU, according to the number of GPUs, a certain number of pivots are selected from the local sorted list. These pivots are broadcast to other GPUs. Then, each GPU sorts all received pivots and selects certain global pivots that should be identical on all GPUs. Based on these global pivots, each GPU's local sorted set is separated into $[n/p]$ partitions.

3. **Partition Exchange.** The *i*th GPU keeps its local *i*th partition and sends others to their corresponding GPUs. All-to-all communication occurs here. GPUDirect is used when multiple GPUs are involved in single-round mode.
4. **Merging Partitions.** Each GPU receives its [*n/p*] partitions from all others and concatenate them together for the output of this stage.

With sorted key-value pairs, MGMR removes all duplicated keys and all values associated with the same keys are stored continuously. Therefore, MGMR can refer each value list through the index of its first value and the list length. All operations are accomplished by GPUs for high performance.

3.5 *Reduce* Stage

Partition Scheduler. *Partition Scheduler* is only used if the input of *Reduce* stage exceeds the sum of all GPUs' memory sizes as in multi-round mode. Fig. 2 shows the detail of how Partition Scheduler works. After *Shuffle* stage, *Partition Scheduler* maintains all value list partitions in data pool on CPU side. The indices of these value lists have been built in advance. When a GPU is idle and its reducer workers come to ask for more *Reduce* work, the *Partition Scheduler* will assign several value lists as a combination with the consideration of load balancing and transfer it over to the designated GPU.

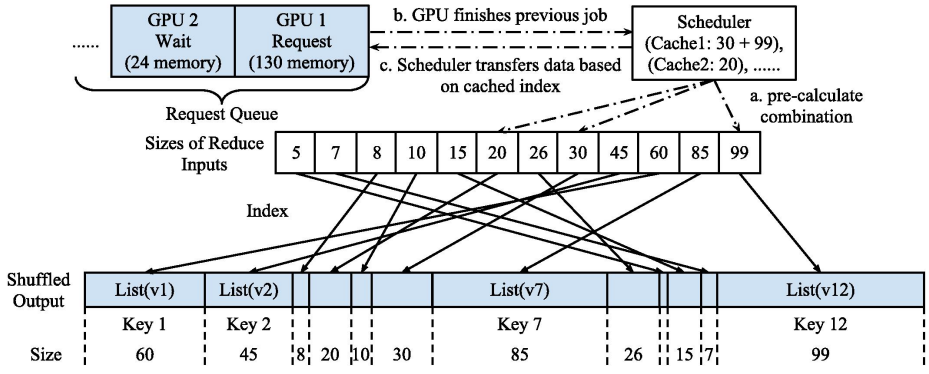

Fig. 2. The interaction between Partition Scheduler and GPUs

An approximate algorithm of subset sum problem from Przydatek 16 is used to estimate how much data can be packed into each GPU memory that will infer the workload. As shown in Fig. 2, while GPUs are busy with their reducers, multiple CPU threads concurrently calculate the possible input combinations for next round GPU execution. When a GPU finishes its work, it can get another one right from *Partition Scheduler*. Self-scheduling is applied for load balancing. However, those CPU threads get jobs ready for GPUs.

Key-Value Processing. In this sub-stage, user-defined reducer function will executed by hardware threads on GPUs. Similar to the situation in *Map* Stage, multiple GPUs' computation and PCIe's bidirectional communication capacities are exploited thoroughly to utilize the advanced features from Nvidia Fermi architecture. Atomic operations are avoided by CUDA version parallel prefix sum where data items are reduced in parallel.

4 Experimental Results

All experiments were conducted on a server containing two Intel Xeon X5660 (2.80GHz, totally 24 cores) with 24 GB RAM and two Nvidia GPUs: Quadro 6000 (1.15 GHz, 5,375 MB global memory, 64KB L1-cache/SM) and Tesla C2070 (1.15 GHz, 5,375 MB global memory, 64KB L1-cache/SM). The server is running the GNU/Linux operating system with kernel version 2.6.32. Testing applications are implemented with CUDA 5.0 and compiled with NVCC compiler in CUDA Toolkit 5.0. CPU versions are implemented with OpenMP using 24 threads to utilize all 24 CPU cores for full capacity.

Two real applications were used for experiments.

4.1 K-Means Clustering

K-Means Clustering (KMC)17 is used in data mining which aims to partition n observations into k clusters where all observations in a cluster are close to the nearest mean. The testing data is randomly generated from a 10k×10k square area with floating-point coordinates. *Map* stages finds the cluster for each point based on means and emit 〈index (cluster), point〉. *Partial Folder* is used to reduce I/O, so only the sum of the x-y coordinates of each cluster is emitted to *Reduce* stage that calculates new means. These three steps are repeated until all means stop changing. Since KMC is NP-hard, we set the test to three rounds for measurable performance. Also, the cluster number k is set to 24 for fair comparison between CPU and GPUs since there are totally 24 CPU cores.

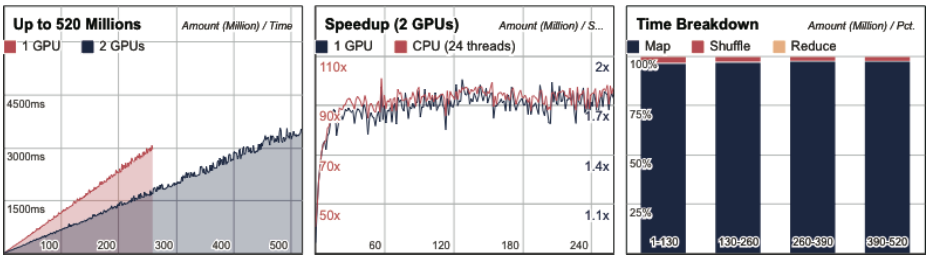

Fig. 3. Experimental results of KMC: execution time, speedup, runtime breakdown

As shown in Fig. 3, we generated up to 520 million points for the experiments. KMC is quite computationally intensive since each point needs to compare with 24 means to find the closest one. Multi-GPU version can declare clear computability advantage. Double-GPU version achieves 91.7 times speed-up over CPU version and 1.7 times speed-up over single-GPU version. *Shuffle* stage takes a small percentage of execution time because of the optimization of *Partial Folder* sub-stage. For the same reason, *Reduce* stage is very light-weighted.

4.2 Unique Phrase Pattern

Unique Phrase Pattern (UPP) can detect the most frequently used phrase patterns. Only two to three-word phrases are acceptable to our tests. The input data is randomly generated from a forty thousand-word dictionary that is pre-hashed. In MGMR, UPP is developed as a three-pass MapReduce. Therefore, no extra work is needed for allocating different sizes of buffers for different phrases. The first two passes count 2-word and 3-word phrases separately. Key-value pairs are emitted as 〈*list* (*hash* (*word1*), ...), *1*〉. Both results are used as the input for the third pass in order to sort all phrases in one mapper for their occurrence. Since UPP originally bounded by I/O, the sub-stage *Partial Folder* in *Shuffle* stage is activated in each pass to reduce the data transfer overhead.

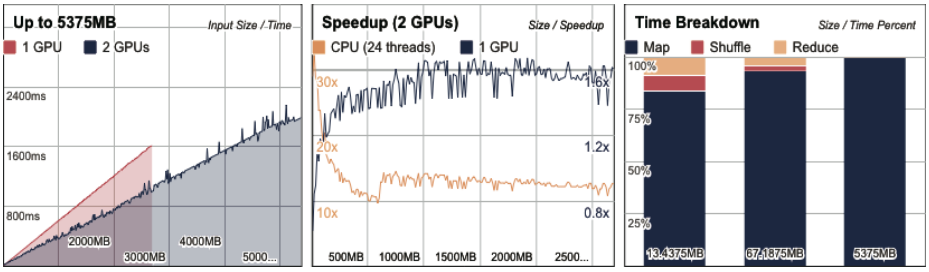

Fig. 4. Experimental results of UPP: execution time, speedup and runtime breakdown

UPP experimental results are shown in Fig. 4. Both GPU versions show very stable efficiency while the problem size increases. Double-GPU version achieves 12.6 times speed-up over CPU version and 1.5 times speed-up over single-GPU version in average. Single-GPU version is only slightly faster than double-GPU version when the input size is very small (less than 45 MB) because of the low GPU occupancy and communication overhead in double-GPU versions' (*Shuffle* stage). According to the runtime breakdown figure, *Map* stage is the most time-consuming portion.

5 Related Work

MapReduce has been implemented on many different platforms such as shared memory system, computer cluster, and GPU workstation. Each implementation has its own contributions and potential issues.

Hadoop 7 MapReduce developed by Apache Software is designed for better programmability in processing vast amount of data in cluster. Hadoop was developed in Java, but Hadoop Streaming allows users to customize their own *Map* and *Reduce* functions in other programming languages such as C and python. Phoenix 8 is a MapReduce implementation for shared-memory systems with multi-core chips and symmetric multiprocessors. Only CPU cores are utilized.

Mars 9 is the first GPU-based MapReduce system. Mars uses an atomic-free output-handling scheme on GPUs, but it has a two-step output process in order to calculate the data allocation and avoid race condition among threads. MapCG 10 designs a memory allocator to reserve buffers in GPU global memory for each warp. However, the atomic operations with global memory cause serious performance penalty. Chen and Agrawal 11 optimized MapCG by executing the Reduce function in GPU shared memory and achieved 2-60 times speedups. GPMR 12 implements MapReduce on GPU clusters to handle big data issue. Partial reductions and accumulation are used to reduce network traffic.

6 Conclusions and Future Work

In this paper, a multi-GPU MapReduce, called MGMR, is developed to tackle with big data issue. Scalability is achieved in both computational power and data size aspects. To avoid the possible communication overhead when multiple GPUs are employed, GPUDirect is applied for inter-GPU interactions without going through CPU memory. Unlike most existing GPU MapReduce systems, MGMR also considers big data input scenario. When data size is larger than the aggregate GPU memory, CPU memory is used to continue the MapReduce operation. Atomic operations are replaced by parallelize one as well. Experimental results have demonstrated MGMR's advantages over both CPU and single-GPU MapReduce in both performance and scalability aspects.

The future work includes extending MGMR to GPU clusters by using RDMA for further performance scalability, integrating it with file systems for fault tolerance, and improving its easy-to-use aspect for programmability.

Acknowledgements. This research is based upon work supported by the Industrial Strategic Technology Development Program (10041971, Development of Power Efficient High-Performance Multimedia Contents Service Technology using Context-Adapting Distributed Transcoding) funded by the Ministry of Knowledge Economy (MKE), Korea, and National Science Council (NSC), Taiwan under grant NSC101-2915-I-126-001, and NVIDIA.

References

1. NVIDIA CUDA Programming Guide 5.0, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
2. OpenCL - The open standard for parallel programming of heterogeneous systems, `http://www.khronos.org/opencl`

3. Caylor, M.: Numerical Solution of the Wave Equation on Dual-GPU Platforms Using Brook+. Presentation, Boise State University (2010)
4. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51(1), 107–113 (2008)
5. Elteir, M., Lin, H., Feng, W., Scogland, T.: StreamMR: An Optimized MapReduce Framework for AMD GPUs. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, pp. 364–371 (2011)
6. Shainer, G., Ayoub, A., Lui, P., Kagan, M., Trott, C., Scantlen, G., Crozier, P.: The development of Mellanox/NVIDIA GPU Direct over InfiniBand a new model for GPU to GPU communications. Computer Science - Research and Development 26(3-4), 267–273 (2011)
7. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Inc./ Yahoo Press (2010)
8. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyraki, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pp. 13–24 (2007)
9. Fang, W., He, B., Luo, Q., Govindaraju, N.K.: Mars: Accelerating MapReduce with Graphics Processors. In: Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, pp. 608–620 (2011)
10. Hong, C.T., Chen, D.H., Chen, Y.B., Chen, W.G., Zheng, W.M., Lin, H.B.: Providing Source Code Level Portability Between CPU and GPU with MapCG. Journal of Computer Science and Technology 27(1), 42–56 (2012)
11. Chen, L., Agrawal, G.: Optimizing MapReduce for GPUs with effective shared memory usage. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, pp. 199–210 (2012)
12. Stuart, J.A., Owens, J.D.: Multi-GPU MapReduce on GPU Clusters. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, pp. 1068–1079 (2011)
13. Alam, S.R., Fourestey, G., Videau, B., Genovese, L., Goedecker, S., Dugan, N.: Overlapping Computations with Communications and I/O Explicitly Using OpenMP Based Heterogeneous Threading Models. In: Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, pp. 267–270 (2012)
14. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for CUDA. In: GPU Computing Gems: Jade Edition, pp. 359–371. Morgan Kaufmann (2011)
15. Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P.S., Shi, H.: On the Versatility of Parallel Sorting by Regular Sampling. Journal of Parallel Computing 19(10), 1079–1103 (1993)
16. Przydatek, B.: A Fast Approximation Algorithm for the Subset-sum Problem. Journal of International Transactions in Operational Research 9(4), 437–459 (2002)
17. Yu, S., Tranchevent, L.-C., Liu, X., Glanzel, W., Suykens, J.A.K., De Moor, B., Moreau, Y.: Optimized data fusion for kernel k-means clustering. Journal of IEEE Transactions on Pattern Analysis and Machine Intelligence 34(5), 1031–1039 (2012)