

Divide-and-Conquer Parallelism for Learning Mixture Models

Takaya Kawakatsu^{1(✉)}, Akira Kinoshita¹, Atsuhiro Takasu²,
and Jun Adachi²

¹ The University of Tokyo, 2-1-2 Hitotsubashi, Chiyoda, Tokyo, Japan
{kat,kinoshita}@nii.ac.jp

² National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda, Tokyo, Japan
{takasu,adachi}@nii.ac.jp

Abstract. From the viewpoint of load balancing among processors, the acceleration of machine-learning algorithms by using parallel loops is not realistic for some models involving hierarchical parameter estimation. There are also other serious issues such as memory access speed and race conditions. Some approaches to the race condition problem, such as mutual exclusion and atomic operations, degrade the memory access performance. Another issue is that the first-in-first-out (FIFO) scheduler supported by frameworks such as Hadoop can waste considerable time on queuing and this will also affect the learning speed. In this paper, we propose a recursive divide-and-conquer-based parallelization method for high-speed machine learning. Our approach exploits a tree structure for recursive tasks, which enables effective load balancing. Race conditions are also avoided, without slowing down the memory access, by separating the variables for summation. We have applied our approach to tasks that involve learning mixture models. Our experimental results show scalability superior to FIFO scheduling with an atomic-based solution to race conditions and robustness against load imbalance.

Keywords: Divide and conquer · Machine learning · Parallelization · NUMA

1 Introduction

There is growing interest in the mining of huge datasets against a backdrop of inexpensive, high-performance parallel computation environments, such as shared-memory machines and distributed-memory clusters. Fortunately, modern computers can have large memories, with hundreds of gigabytes per CPU socket, and the memory size limitation may not continue to be a severe problem in itself. For this reason, state-of-the-art parallel computing frameworks like Spark [1, 2], Piccolo [3], and Spartan [4] can take an *in-memory* approach that stores data in dynamic random access memory (DRAM) instead of on hard disks. Nonetheless, there remain four critical issues to consider: *memory access speed*, *load imbalance*, *race conditions*, and *scheduling overhead*.

A processor accesses data in its memory via a bus and spends considerable time simply waiting for a response from the memory. In shared-memory systems, many processors can share the same bus. Therefore, the latency and throughput of the bus will have a great impact on calculation speed. For distributed-memory systems in particular, each computation node must exchange data for processing via message-passing frameworks such as MPI¹, with even poorer throughput and greater latency than bus-based systems. Therefore, we should carefully consider memory access speeds when considering the computation speed of a program. The essential requirement is to improve the reference locality of the program.

Load imbalance refers to the condition where one processor can be working hard while another processor is waiting idly, which can cause serious throughput degradation. In some data-mining models, the computation cost per observation data item is not uniform and load imbalance may occur. To avoid this, dynamic scheduling may be a solution.

Another characteristic issue in parallel computation is the possibility of race conditions. For shared-memory systems, if several processors attempt to access the same memory address at the same time, the integrity of the calculation can be compromised. Mutual exclusion using a semaphore [5] or mutex can avoid race conditions, but can involve substantial overheads. As an alternative, we can use atomic operations supported by the hardware. However, this may remain expensive because of latency in the cache-coherence protocol, as discussed later.

The fourth issue is scheduling overhead. The classic first-in-first-out (FIFO) scheduler supported by existing frameworks such as OpenMP² and Hadoop³ is implemented under a *flat partitioning* strategy, which divides and allocates tasks to each processor without detailed consideration of their interrelationships. A flat scheduler cannot adjust the granularity of the subtasks and it tends to allocate tasks with extremely small granularity. Because a FIFO scheduler has only one task queue and all processors access the queue frequently, the queuing time may become a serious bottleneck, particularly with fine-grained parallelization.

In this paper, we propose a solution for these four issues by bringing together two relevant concepts: *work-stealing* [6,7] and the *buffering solution* under a recursive divide-and-conquer-based parallelization approach called *ADCA*. The combination of a work-stealing scheduler with our ADCA will reduce scheduling overheads because of the absence of bottlenecks, while ADCA also achieves efficient load balancing with optimum granularity. Buffering is a method whereby each processor does local calculations wherever possible, with a master processor integrating the local results later. This helps to avoid both race conditions and latency caused by the cache-coherence protocol. ADCA and the buffering solution are our main contributions.

As target applications for ADCA, we focus on machine-learning algorithms that repeat a learning step many times, with each step handling the observation data in parallel. Expectation-maximization (EM) algorithms [8,9] on

¹ <http://www.mpi-forum.org>.

² <http://www.openmp.org>.

³ <http://hadoop.apache.org>.

a Gaussian mixture model (GMM) or a hierarchical Poisson mixture model (HPMM) [10, 11] are well-known examples of such applications. Mixture models are popular and versatile; their applications include wireless sensor networks [12, 13], speech recognition [14, 15], and moving object detection [16–19]. Another principal application, back-propagation-based learning [20] of neural networks, can also be parallelized using the same approach [21, 22].

In Sect. 2, we formulate parallel computing in general terms, introducing our main concept, *three-step parallel computing*, and then introduce work-stealing and the buffering solution. In Sect. 3, we summarize related work on parallel EM algorithms and then explain our EM algorithm based on ADCA. In Sect. 4, we demonstrate our method’s superior scalability to FIFO scheduling and to the atomic solution by experiments with GMMs. We also demonstrate our method’s robustness against load imbalance by experiments with HPMMs. Finally, we conclude this paper in Sect. 5.

2 Parallel Computation Models

There are a vast number of approaches to parallel computing; it is not easy for users to select an approach that meets their requirements. Even though parallel technologies may not seem to cooperate with each other, we can integrate them according to the *three-step parallel computing* principle, which contains three phases: *parallelization*, *execution*, and *communication*. In the parallelization phase, the programmer writes the source code specifying those parts where parallel processing is possible. In the execution phase, a computer executes the program serially, assigning tasks to its computation units as required. Finally, in the communication phase, the units synchronize and exchange values.

2.1 Parallelization of Algorithms

In the parallelization phase, the programmer effectively informs the computer which statements can be executed in parallel. This can be separated into two subphases: the *algorithm phase* and the *directive phase*. In the algorithm phase, the programmer chooses the form of parallelism: *data parallelism* [23] or *task parallelism*. The programmer then specifies the parallelizable statements in the directive phase. For data parallelism, the program is described as a loop, as illustrated in Fig. 1a. That is also called *loop parallelism*. OpenMP supports loop parallelism by the directive `parallel for`. Furthermore, single-instruction multiple-data (SIMD) [24] instructions, such as Intel streaming SIMD extensions, can be categorized as data parallelism. When exploiting data parallelism, we must assume that the program describes an operator that takes an array element as an argument.

Next, task parallelism can be described by using `fork` and `join` functions in a recursive manner, as illustrated in Fig. 1b. After the `fork` function is called, a new thread is created and executed. Each thread processes a user-defined task, and the calculation result is returned to the invoker thread by calling the `join`

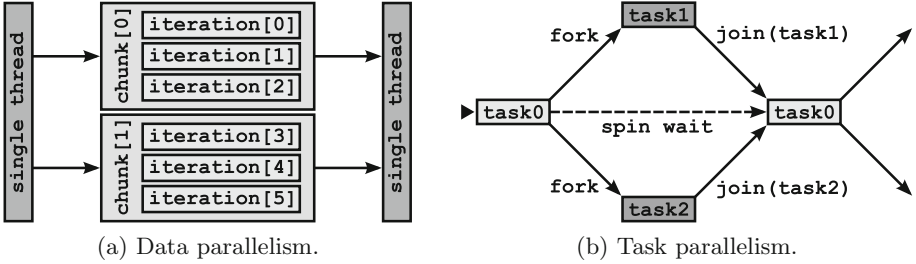


Fig. 1. Data parallelism and task parallelism. A parallel program can be described in a loop manner or a fork-join manner.

function. Actually, the `fork` and `join` functions are provided by the pthreads, `pthread_create` and `pthread_join`, respectively.

In many cases, the critical statement that has the most significant impact on the execution time is a `for` loop with many iterations. A data-parallel program can be much simpler than a task-parallel program. For this reason, parallel loops are frequently exploited in computationally heavy programs. The EM algorithm on a GMM can be parallelized in the loop manner [25–30]. However, parallel loops are not applicable when the data have mostly nonarray structures like graphs or trees. The HPMM is a simple example of such a case. Therefore, parallelizable machine learning for graphical models must be described in a *fork-join* manner.

In practice, data and task parallelism can work together in a single program, such as forking tasks in a parallel loop or exploiting a parallel loop in a recursive task, because parallel loops can be treated as the syntactical sugar of the `fork` and `join` functions. Of course, there are devices that hardly support task parallelism, such as graphical processing units (GPUs). Task parallelism on a GPU remains a challenging problem [31, 32].

Finally, the directive phase can be categorized as involving *explicit directives* or *implicit directives*. The `fork` and `join` functions are examples of explicit directives that permit programmers to describe precisely the relationships between forked tasks. For the implicit case, a scheduler determines automatically whether statements are to be executed in parallel or serially. That decision is realized on the assumption that each task has referential transparency. That is, there are no side effects such as destructive assignment to a global variable.

2.2 Parallel Execution Mechanism

A program that manages tasks is called a *scheduler*, dealing with three subphases: *traversal*, *delivery*, and *balancing*.

In the traversal phase, the scheduler scans the remaining tasks to determine the order of task execution. In task parallelism, tasks have a recursive tree-based structure, and in general, there are two primary options, depth-first traversal, or breadth-first traversal.

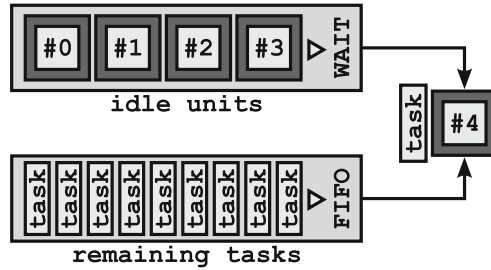


Fig. 2. General FIFO-based solution to counter load imbalance. The program is partitioned into tasks that are allocated one by one to the computation units.

Then, in the delivery phase, the scheduler determines the computation unit that executes each task. This phase plays an important role in controlling reference locality, with the scheduler aiming to reduce load-and-store latency by allocating each task to a computation unit located near the data associated with that task. This is particularly important for machine-learning algorithms, where the computer must repeat a learning step many times until the model converges. Ideally, the scheduler should assign tasks to the computation units so that each unit handles the same data chunk in every learning step, reducing the necessity for data exchanges between units. However, such an optimization does not make sense if the program then has serious load imbalances. In some machine-learning algorithms, the computation cost per task may not be uniform.

In the balancing phase, the scheduler relieves a load imbalance when it detects an idling computation unit. This is an *ex post* effort, whereas the delivery phase is an *ex ante* effort. There are two options for this phase: *pushing* [33–35] and *pulling* [6, 7, 36–40]. Pushing is when a busy unit takes the initiative as a producer and sends its remaining tasks to idling units by passing messages whenever requested by the idling units. In contrast, pulling is when an idle unit takes the initiative as a consumer and snatches its next task from another unit.

The FIFO scheduling illustrated in Fig. 2 is a typical example of a pulling scheduler. An idling unit tries to snatch its next task for execution from a shared task queue called the *runqueue*. The program is partitioned into many subtasks that are appended to the runqueue by calling the `fork` function. Because of its simplicity, FIFO scheduling is widely used in Hadoop and UNIX. While this may appear to be a good solution, it can cause excessively fine-grained task snatching and the resulting overhead will reduce the benefits of the parallel computation. A shared queue is accessed frequently by all computation units and therefore behaves as a single point of failure. Hence, the queuing time may become significant, even though the queue implementation utilizes a lock-free-based protection technique instead of mutual exclusion such as a mutex. To avoid this, the task partitioning should be as coarse-grained as possible; however, load balancing will then be less effective. As another issue, we suspect that the ability

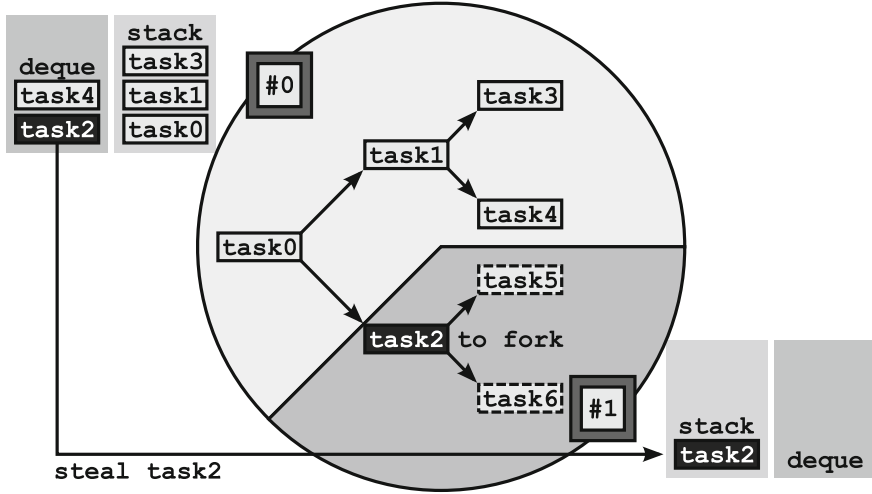


Fig. 3. Breadth-first task distribution with a work-stealing scheduler. Idle unit #1 steals a task in a FIFO fashion to minimize the stealing frequency.

to tune referential locality can be poor and this will become a serious problem, particularly in the context of distributed processing.

Mohr et al. introduced a novel balancing technique called *work-stealing* for their LISP system [6, 41]. They focused on the property of a recursive program that the size of each task can be halved by expanding the tree-structured tasks. As illustrated in Fig. 3, the work-stealing scheduler first expands the root task into a minimum number of subtasks, and distributes them to computation units either by pushing or pulling. When a computation unit becomes idle, the scheduler divides another unit's task in half and reassigns one half to the idle unit. This behavior is called work-stealing. In this way, the program is always divided into the smallest number of tasks required, thereby achieving a minimum number of stealing events.

A typical work-stealing scheduler [35, 40, 42] is constructed by exploiting a thread library such as pthreads. Each computation unit is expressed as a *worker thread* fixed to the unit and has its own local *deque* or double-ended queue to hold tasks remaining to be executed. Tasks are popped by the owner unit and executed one by one in a last-in first-out (LIFO) fashion. When there are no idling units, each worker behaves independently of the others. If a unit becomes idle, with an empty deque, the unit scouts around other units' deques until it finds a task and steals it in a FIFO fashion, as described in Algorithm 1.

Of course, a remaining task may create new tasks by calling a `fork` function, and such subtasks are appended into the local deque in a LIFO fashion, as shown in Algorithm 1. Hence, the tasks in each deque are stored in order of descending age. That is, an idling unit steals the oldest remaining task, and that will be the one nearest to the root of the task tree. This is the reason for the work-stealing

Algorithm 1. A worker thread’s behavior in a work-stealing scheduler.

Require: *myself*: the worker thread, *victim*: another worker thread

```

procedure FORK(function, arguments)
    task = new task(function, arguments)
    myself.deque.append(task)
    return task
end procedure
procedure JOIN(task)
    repeat
        if myself.deque.is_empty then
            next = victim.deque.pop_FIFO()
            next.execute()
        else
            next = myself.deque.pop_LIFO()
            next.execute()
        end if
    until task.is_finished
end procedure

```

scheduler being able to achieve the minimum number of stealing events necessary. In addition, there is no single point of failure, and the overhead will be smaller than that for the FIFO scheduler.

2.3 Communication Mechanism

In the communication mechanism, each computation unit exchanges values via a bus or network. For example, when calculating the mean value of a series, each unit will calculate the mean of a chunk, with a master unit then unifying the means into a single value. There are two options for the communication mechanism: *distributed memory* and *shared memory*.

In a distributed-memory system, each computation unit has its own memory and its address space is not shared with other units. Communication among units is realized by explicit message passing, with greater latency than local memory access.

In a shared-memory system, several computation units have access to a large memory with an address space shared among all computation units. There is no need for explicit message passing, with communication achieved by reading from or writing to shared variables.

A great problem for shared-memory systems is the possibility of *race conditions*, as shown in Fig. 4a. Suppose that two computation units, #0 and #1, are adding some numbers into a shared variable `total` concurrently. Such an operation is called a *load-modify-store* operation, but the result can be incorrect, because of conflicts between loading and storing. Using atomic operations can be a solution. An atomic operation is guaranteed to exclude any load or store operations by other computation units until the operation finishes.

Note that, because memory access latency suspends an operation for a time, modern processors support the *out-of-order* execution paradigm, going on to

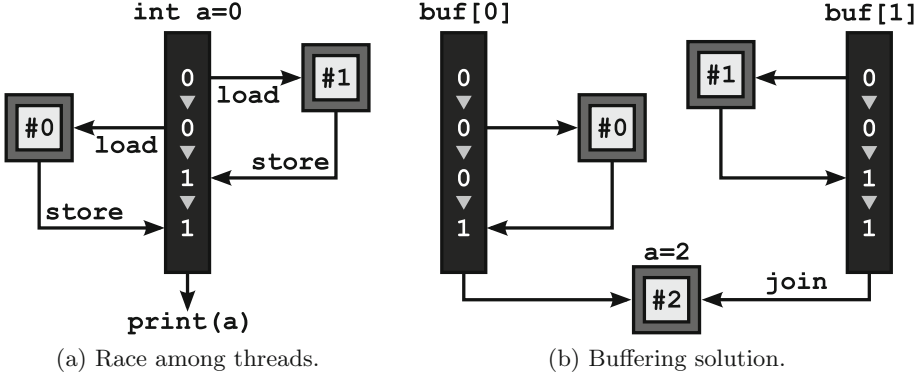


Fig. 4. A race condition among computation units and our buffering solution. The sum may be incorrect if several units access the same variable at the same time.

execute other instructions until the processor obtains all required data from the memory. This accelerates serial program execution, but can compromise the integrity of a parallel program that includes some critical instructions that must be strictly executed in a particular order. Figure 5 is an example of such a program, where `thread2` waits until `thread1` updates `value`. The loop is called *spin waiting* or *busy waiting*, and is frequently used for thread synchronization. If the statements are executed out of order, `thread2` may load an old value before the update by `thread1`. As a solution, an atomic operation often involves a *memory fence*, instructions that inhibit out-of-order execution.

Considering the memory access patterns of machine-learning algorithms, the use of atomic operations may not be an adequate solution. Main memories based on DRAM operate more slowly than processors, so that modern computers insert caches based on static random-access memory (SRAM) between the processors and main memories. Main memory access is blocked if the cache memory has a valid replica of the addressed data. The problem is that several computation units may have replicas of the same data item in their own cache memories and the stored replica values may become outdated. Machine-learning algorithms access all the observation items simultaneously to calculate summations in each learning step, and cache conflicts may occur frequently. A *cache-coherence* [43, 44] protocol invalidates the old replica to relieve conflicts, as illustrated in Fig. 6. However, this may become a serious bottleneck. For this reason, we recommend a *buffering solution*, as shown in Fig. 4b. The solution separates the memory addresses physically to avoid cache conflicts. Each computation unit calculates a summation into its local buffer and the master unit retrieves these to calculate the total summation after calling a `join` function. This method can be combined easily with our divide-and-conquer-based parallelization approach.

2.4 Parallel Computing Frameworks

Many frameworks assist parallelization, parallel execution, and communication.

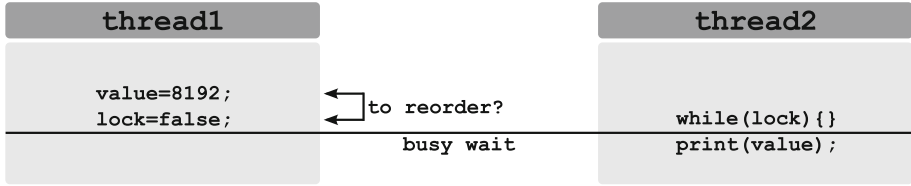


Fig. 5. Data exchange between threads by spin waiting. The synchronization will fail if the statements are executed out of order.

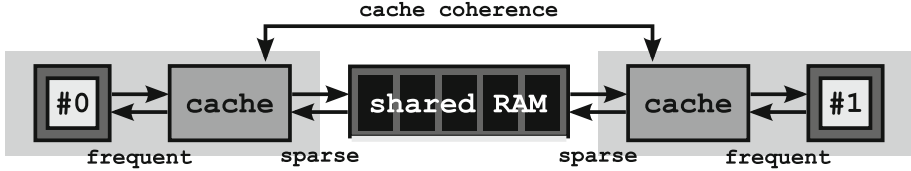


Fig. 6. Cache coherence among computation units. The protocol will cause a bottleneck whenever units are accessing the same address.

As a parallelization framework, we could use a high-performance-computing (HPC) language such as Cilk [7, 36], X10 [45], or Chapel [46]. They support task parallelism, and Chapel also supports data parallelism with sophisticated syntax. OpenMP is another well-known framework for task and data parallelism for shared-memory environments. These languages and libraries are examples of the explicit-directive approach, while MapReduce [47] is a library-level variant of the implicit-directive approach.

As a parallel execution framework, we could use a work-stealing scheduler such as Intel’s TBB⁴, qthreads [38, 39], or MassiveThreads [40]. Pthreads⁵ does not support work-stealing by itself, but is an essential component for implementing such schedulers.

In the context of communication mechanisms, some general-purpose languages provide helpful programming models for parallel computation, even though they were not designed primarily as HPC languages. For example, Go⁶ supports sophisticated syntax for message passing. In low-level programming, MPI is a standardized message-passing framework that supports `MPI_Send` and `MPI_Recv`. In the context of enterprise applications, Hadoop is a well-maintained platform for distributed data processing. In the context of shared-memory communication, the simplest example is multithread programming using a thread library. In C++11, all global variables are shared among threads by default, unless using a `thread_local` specifier, and by using `std::atomic` templates⁷, a programmer can write thread-safe access to shared data.

⁴ <http://www.threadingbuildingblocks.org>.

⁵ <http://computing.llnl.gov/tutorials/pthreads/>.

⁶ <http://golang.org>.

⁷ <http://www.cplusplus.com/reference/atomic/atomic>.

The ADCA proposal in Sect. 3.2 follows the principles of parallelization, parallel execution and communication. In the parallelization phase, ADCA adopts task parallelism to describe a divide-and-conquer algorithm. In the parallel execution phase, ADCA utilizes a pulling-based work-stealing scheduler to distribute tasks to computation units. Then, ADCA realizes communication among units by using shared-memory; the cache coherence problem is reduced thanks to the buffering solution.

3 Parallel EM Algorithms

The EM algorithm comprises an *E-step* and an *M-step*. The E-step computes a single posterior $P(k|\mathbf{x}_n)$ for each pair of an observation item \mathbf{x}_n and a mixture component k that indicates how likely it is that the item \mathbf{x}_n was generated by the component. In the M-step, the posteriors are summed to estimate revised parameter values. The E-step is then repeated using the revised model. This EM iteration continues until the likelihood function \mathcal{L} , which indicates how well the model regenerates the dataset, converges to a maximum.

To parallelize the EM algorithm, we should divide each E-step and M-step into a number of tasks and allocate them to computation units. For GMMs, the axis of the observation item \mathbf{x}_n and the axis of mixtures k are available for this division. As illustrated in Fig. 7, reference locality is maximized whenever we divide the posterior table into squares because the total number of observation items and model parameters to be loaded to calculate the posterior subtable is minimized. The number of observation items is usually much greater than the number of mixture components. Consequently, we must divide the n axis more than the k axis.

The EM algorithm includes summation processes in the M-step, and we must take measures against race conditions among computation units. In general, race conditions are resolved by using mutual exclusion techniques such as semaphores, or by using atomic instructions. However, we do not recommend these approaches because mutual exclusion involves a blocking time, and an atomic operation on a shared variable involves the cache-coherence problem. The best solution is to let each computation unit use its own local memory to hold intermediate results, thereby computing a partial sum independently, before a single unit retrieves the intermediate results to compute the total.

3.1 Related Work

Nonuniform memory access (NUMA) is a shared-memory architecture for which a computation unit and local memory form a pair called a *NUMA node*. Modern processors support NUMA at the chip level and a NUMA node is generally equivalent to a processor socket. The memory address space is continuous, enabling each NUMA node to access another node's local memory in the same way as for its own local memory. In a shared-memory system, all computation units share the threads of a process and any unit can execute a thread. Because the threads

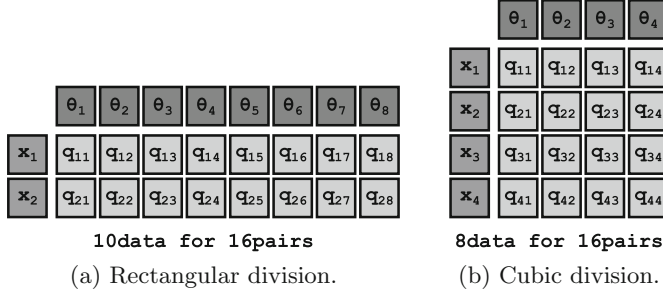


Fig. 7. Cubic division of the posterior table. Space requirements are minimized when the posterior table is divided into cubes. The symbols such as θ and q_{nk} are described in detail in Appendix A.

share a common address space assigned to the process, we can implement programs without explicit message passing among threads. However, noting that all NUMA nodes will be interconnected via a bus, nonlocal memory accesses will have higher latency than local memory accesses for a computation unit. Consequently, programmers should aim to maximize reference locality to increase the proportion of local memory accesses. Kwedlo [25] proposed a parallel version of the EM algorithm for a NUMA computer. He used the parallel loop of OpenMP and introduced two techniques for improving the reference locality, the buffering solution and *first touching*.

When we parallelize an EM algorithm, we normally partition the observation items into a number of data chunks and assign them to threads. On the one hand, a race condition will never arise in the E-step because there is no summation over the observation items. On the other hand, parameter recalculation in the M-step requires summation over the posteriors. Whenever several computation units read from and write to the same address *simultaneously*, a race condition is possible, with the revised parameters differing from the correct values. To avoid this, Kwedlo arranged an independent array for each unit, with each unit calculating a partial sum into its own array. The partial sums are then integrated by a single thread at the end of the M-step. He introduced a buffering solution, although OpenMP supports safe summation via the `reduction` clause, because that clause cannot handle array types [25]. Accordingly, Kwedlo proposed his own buffering solution, coincidentally similar to Fig. 4b.

First-touching [48, 49] is a well-known optimization method in the context of combining Linux⁸ and NUMA. In Linux, a logical memory address is not bound to a physical memory address initially. When a thread accesses the logical address for the first time, a small physical memory space called a *page*, is selected from the closest NUMA node and allocated to the logical address. Therefore, Kwedlo made all threads access their own chunk before running the EM algorithm [25].

⁸ <https://www.kernel.org>.

Distributed memory is a parallel clustering-based architecture that comprises many computers called *nodes* interconnected via a network. The meaning of *node* in distributed computing is somewhat different from that used in the description of NUMA. For distributed memory, each node is a processor with several cores that use a shared-memory architecture. That is, the distributed-memory system has at least two levels of memory hierarchy: *internode* and *intranode*. Internode communication is realized by explicit message passing, with internode latency being greater than intranode shared-memory access. Therefore, we must consider reference locality more carefully than for shared-memory systems.

The message-passing communication model is applicable to both distributed and shared-memory systems. It rarely depends on the detailed architecture, with its application being wider than that of the shared-memory programming model. For this reason, the message-passing model is more popular with programmers using high-level languages such as Java and Scala. A programmer can use highly abstract concurrent-execution models such as MapReduce, with the background scheduler then assigning tasks to the computation units. MapReduce, supported by Hadoop, offers a simple but powerful abstraction. However, that is too abstract to enable control of the reference locality, unlike NUMA programming exploiting the first-touch policy. Therefore, MapReduce might be convenient but can result in poor throughput. Its handling of hard-disk I/O overhead is another principal reason for its poor performance [1].

Currently, there are several parallel-computing frameworks based on message passing, such as Spark [1], Piccolo [3], and GraphLab [50,51]. Spark is a framework that aligns data in memory to reduce hard-disk I/O overheads, and provides its own distributed, immutable collection framework called the resilient distributed dataset (RDD) [2]. Because RDD elements are in memory, Spark runs faster than Hadoop MapReduce, which must read observation dataset from hard disks each time they are required. Piccolo is a distributed in-memory hash-table framework that runs parallel applications with high efficiency, similarly to RDD. GraphLab is a distributed machine-learning framework in which the programmer describes calculations and data flows by using directed graphs. Each node behaves as if it was a local Map or Reduce facility, with the many Map and Reduce operations all running in parallel.

In the world of low-level programming, *hybrid parallel computing* [52–54] is a popular approach. It uses a thread implementation such as pthreads inside the nodes and MPI among the nodes. Yang et al. [26] proposed an EM algorithm using a hybrid parallelization approach. It divides and conquers the observation items and integrates partial sums at the end of the M-step, as shown in Fig. 8. The implementation has a hierarchical structure. First, the master node assigns an observation data subset to each distributed node by utilizing MPI. Next, each distributed node partitions its subset into smaller subsets and allocates them to threads running on each node. At the end of every M-step, each distributed node calculates its internal summation and the master node collects them to calculate the total sum. This approach achieves good reference locality because nodes do not exchange any values until the intranode calculations are completely finished.

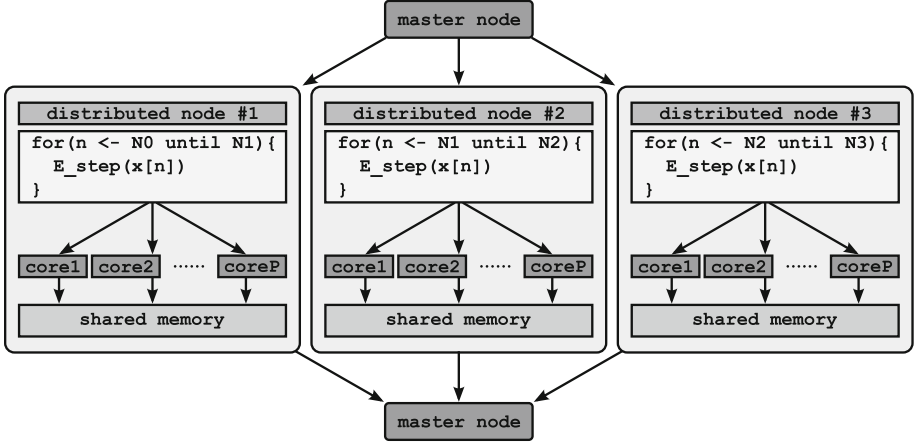


Fig. 8. Hybrid message-passing and shared-memory parallelization. This approach uses a thread inside the node and message passing among the nodes.

The authors attribute the reduction in scheduling overheads to *static scheduling*. That is, their approach divides the data into equal-sized subsets before running the EM algorithm and never deals with load balancing. Of course, the processing throughput might suffer if load imbalances occur.

3.2 Our ADCA Proposal

To utilize a work-stealing scheduler, we must transform the EM algorithm to a divide-and-conquer form, which is not difficult because we can divide the observation dataset recursively and calculate the posteriors in parallel in the E-step. In the M-step, we repartition the dataset to recalculate the parameters recursively as shown in Algorithm 2. This version of the EM algorithm performs effective dynamic load balancing.

ADCA also achieves good reference locality. The divided observation subset and its counterpart will be aligned closely in the memory address space. Machine-learning algorithms repeat learning steps until the model regenerates the training data. Therefore, the programmer can optimize ADCA so that each computation unit retrieves a chunk of the dataset from storage before processing, and handles only its own local chunk in every subsequent step. This optimization may reduce internode I/O transactions dramatically in distributed-memory systems, which is a direction for our future work.

Note that the observation items should not be divided into single data items because the calculation cost per observation item would then be very small, and too-frequent task-switching operations might degrade the processing throughput. Therefore, we introduce a *grain size* parameter as the minimum size for a subset of the observation data. When recursive division of the subsets reaches the grain size, no further division occurs. Although we do not examine selection methods

Algorithm 2. Divide&conquer-based EM algorithm for a GMM.**Require:** \mathbf{x}_n : observation items, N : number of observation items, *grain*: grain size**Ensure:** w_k : weight, μ_k : mean, S_k : covariance

```

repeat
  Estep( $\mathbf{x}_1, \dots, \mathbf{x}_N$ )
  Mstep( $\mathbf{x}_1, \dots, \mathbf{x}_N$ )
until likelihood converges

procedure ESTEP(chunk of  $\mathbf{x}_n$ )
  if chunksize > grain then
    Estep(half of  $\mathbf{x}_n$ )
    Estep(half of  $\mathbf{x}_n$ )
  else
    for each pair  $(\mathbf{x}_n, k)$  do
      calculate  $q_{nk}$  as  $P(k|\mathbf{x}_n)$ 
    end for
  end if
end procedure

procedure MSTEP(chunk of  $\mathbf{x}_{snk}$ )
  if chunksize > grain then
    task1 = Mstep(half of  $\mathbf{x}_n$ )
    task2 = Mstep(half of  $\mathbf{x}_n$ )
     $sum^1$  = join task1
     $sum^2$  = join task2
    return  $sum^1 + sum^2$ 
  else
     $sum = 0$ 
    for each pair  $(\mathbf{x}_{sn}, k)$  do
       $sum_k += (q_{nk}, q_{nk}\mathbf{x}_n, q_{nk}\mathbf{x}_n^2)$ 
    end for
    return  $sum$ 
  end if
end procedure

```

for the grain size here, it can be smaller than for the FIFO scheduler because of the small overhead of the work-stealing-based scheduler, as described in Sect. 2.2.

ADCA has another advantage, the avoidance of race conditions. As shown in Algorithm 2, the parameter recalculation in the M-step is implemented using buffering, and there are no critical sections. Accordingly, there is no need for mutual exclusion or atomic operations on shared variables, which enables much faster computation. Of course, our approach may have the disadvantage of requiring more memory than other approaches do.

4 Experiment and Results

Table 1 describes the experimental environments. For most experiments, we used the `hu080`, but the strong and weak scaling of ADCA were also measured using the `hp160`. Both are shared-memory computers with many NUMA nodes, using CentOS⁹ and the GNU compiler collections (`gcc`¹⁰) 5.2 and 4.8.

To demonstrate the scalability and robustness against load imbalance of our approach, we transformed the EM algorithms into the divide-and-conquer form shown in Algorithm 2. In a previous paper [55], we described the implementation of ADCA in the programming language Chapel, employing MassiveThreads [40] as the work-stealing scheduler, and compared it with a FIFO approach that used OpenMP. For the purposes of this paper, we have implemented a work-stealing scheduler and a FIFO scheduler in C++11 to unify the experimental conditions,

⁹ <http://www.centos.org>.

¹⁰ <http://gcc.gnu.org>.

Table 1. Experimental machine environments.

Name	hu080		hp160	
CPU	Xeon	E7 4870	Xeon	E7 8891 v2
	Clock	2.4 GHz	Clock	3.2 GHz
	Cores	10	Cores	10
Cache	L1d	32 kB/core	L1d	32 kB/core
	L2	256 kB/core	L2	256 kB/core
	L3	30.0 MB	L3	37.5 MB
NUMA	Nodes	8	Nodes	16
	RAM	64 GB/node	RAM	0.75 TB/node

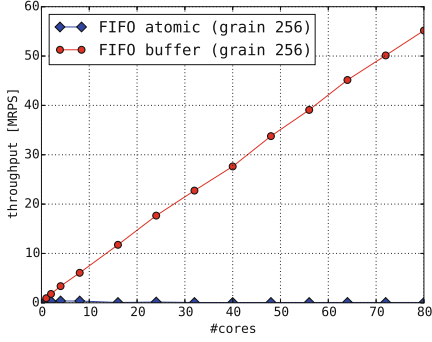
excluding the schedulers and parallelization approaches. Both schedulers employ the lock-free deque implementation proposed by Arora et al. [56], instead of using a mutex-based deque such as MassiveThreads. We examined three aspects of the system: the effect of the buffering solution, robustness against fine-grained parallelism, and robustness against load imbalance. The effect of the buffering solution was ascertained by comparison with the atomic solution using a GMM. Robustness against fine-grained parallelism was tested by comparison with the FIFO approach, again using the EM algorithm on a GMM. Robustness against load imbalance was examined by learning both load-imbalanced and load-balanced HPMM datasets.

For the experiments, we prepared randomly generated training data. For the GMMs, each observation item was generated by eight mixture components, and was expressed as an eight-dimensional vector, with each value expressed in 64-bit floating-point form. For the HPMMs, each item was generated by eight mixture components, and was expressed as a two-dimensional 64-bit integer vector.

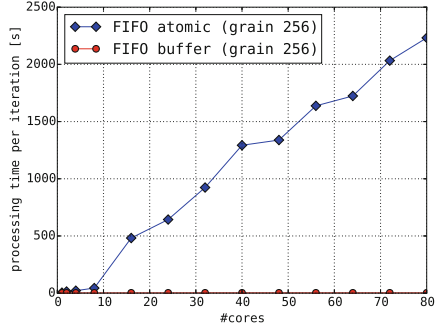
The graphs below show results for both *strong* and *weak scaling*. They indicate how processing speed varies with the number of computation units, but the size of the dataset is fixed in strong scaling, whereas the size varies in proportion to the number of units in weak scaling. For strong scaling, the vertical axis indicates the throughput in megarecords per second (MRPS), whereas it indicates the processing time per EM iteration for weak scaling.

4.1 Effect of Buffering Solution

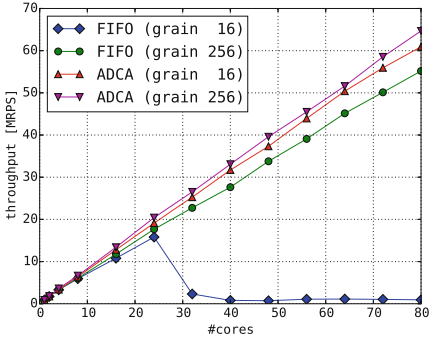
To demonstrate the effect of the buffering solution, we compared the difference in scalability between the buffering solution and the atomic solution for the EM algorithm on a GMM. We adopted the FIFO approach in both cases and set the grain size to 256. That is, each task handles 256 observation items. The datasets had 268,435,456 items in total in strong scaling and 2,097,152 items per core in weak scaling. Figure 9 shows the comparison results. The atomic solution did not so much speed up as slow down in weak scaling, whereas the buffering solution achieved an almost linear speedup. The atomic solution decelerated at a rate



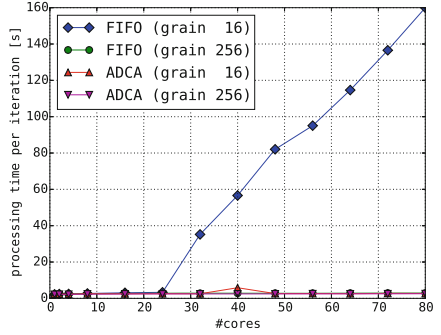
(a) Strong scaling.



(b) Weak scaling.

Fig. 9. Atomic solution vs buffering solution (hu080).

(a) Strong scaling.



(b) Weak scaling.

Fig. 10. Scalability of the EM algorithm on a GMM (hu080).

of 28 seconds per core. Note that the total throughput was invariant regardless of the number of cores. This suggests that there exists a bottleneck setting the upper limit of the throughput. We suspect that the cache-coherence protocol was the main factor.

4.2 Robustness Against Fine-Grained Parallelism

To demonstrate the robustness against fine-grained parallelization, we compared the difference in scalability between the FIFO-based approach and ADCA, while varying the grain size from 256 items to 16 items. The sizes of the datasets were the same as those for Fig. 9. Figure 10 shows the evaluation result. When the grain size was set to 256, the FIFO approach accelerated between 1 and 80 cores at a rate that was 15.7% less than that for ADCA. When the grain size was set to 16, the FIFO approach decelerated beyond 24 cores, and it did not speed up beyond a factor of 17.5. In contrast, our approach achieved a near-linear speedup in both cases. This could be explained by the overhead of the shared runqueue.

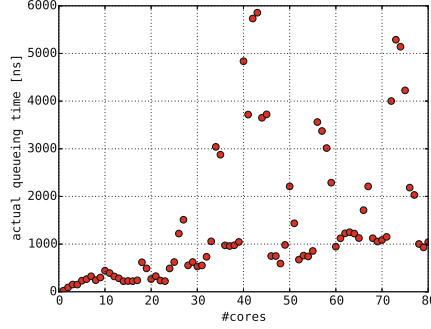
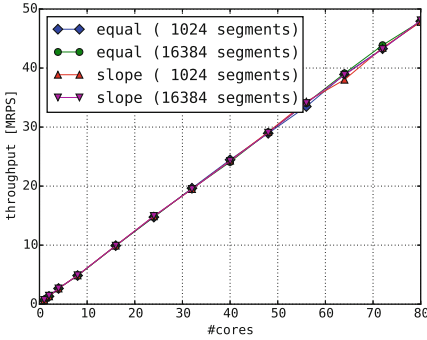
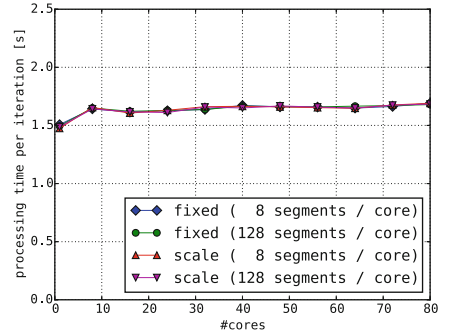


Fig. 11. Maximum queuing time with FIFO scheduling (hu080).



(a) Strong scaling.



(b) Weak scaling.

Fig. 12. Scalability of the EM algorithm on an HPMM (hu080).

To test that hypothesis, we also measured the queuing time of the shared runqueue. The queuing time was shorter than a millisecond, which made direct measurements difficult. We therefore implemented a program that repeats task-popping from a shared queue 268,435,456 times to calculate the average queuing time. Figure 11 shows the result. The queuing time increased as the number of cores increased. That is, a popping request was cancelled when several computation units simultaneously tried to obtain a task from the queue, thanks to the protection technique proposed by Arora et al. [56]. As seen in Fig. 11, the queuing required several micro-seconds whenever a queuing rush occurred.

As seen in Fig. 10, a single core could handle 0.9 observation items per microsecond, which means that the queuing time has a great impact on the lack of acceleration. For the FIFO case, the shared runqueue was accessed frequently by all computation units. For the ADCA, such rushes are rare. This is the reason for the superior robustness against fine-grained parallelization.

4.3 Robustness Against Load Imbalance

To demonstrate the robustness against load imbalance of ADCA, we evaluated strong and weak scaling using the EM algorithm on an HPMM. For the strong scaling, we tested two datasets: **equal** and **slope**. In the **equal** dataset, each segment had the same number of observation items. In the **slope** dataset, the number of observation items in each segment was made proportional to the segment ID, assigned continuously from 1 to either 1024 or 16,384. The datasets comprised 134,217,728 observation items. The grain size was set to 256. For weak scaling, we tested two series of datasets: **fixed** and **scale**. In the **fixed** datasets, the number of segments was constant, regardless of the number of cores. In the **scale** datasets, the number of segments was proportional to the number of cores. The datasets contained 1,048,576 items per core and the grain size was set to 256. Figure 12 shows the results. For both the **equal** and **slope** datasets, ADCA achieved an almost linear speedup. Note that the graphs almost exactly

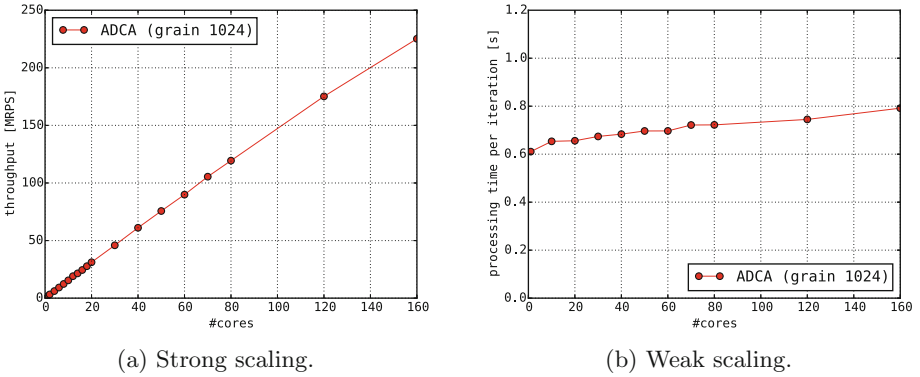


Fig. 13. Scalability of the EM algorithm on a GMM (hp160).

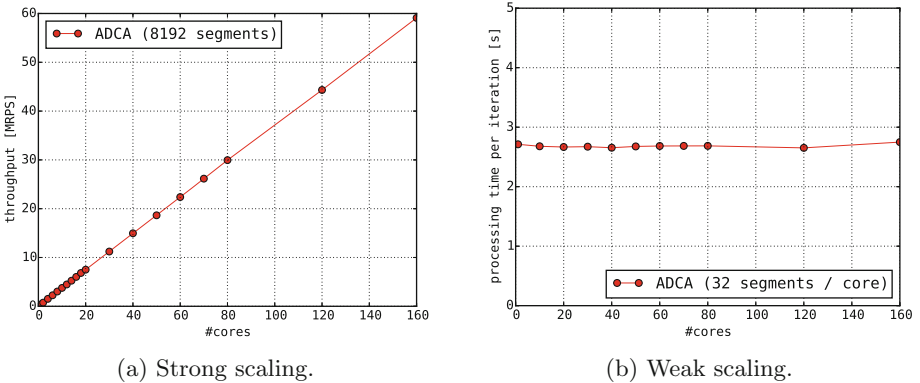


Fig. 14. Scalability of the EM algorithm on an HPMM (hp160).

match each other, with load imbalance having little influence. The weak scaling results demonstrate the great flexibility of ADCA, which is applicable to both *large* tasks handling many observation items and to *small* tasks handling a few observation items. In all cases, the throughput was constant and the processing time was determined only by the size of the dataset.

4.4 Scalability on a 160-Core NUMA Machine

We evaluated strong and weak scaling for our method not only on **hu080**, but also on **hp160**. The GMM datasets involved a total of 268,435,456 observation items for strong scaling and 1,048,576 observation items per core for weak scaling. The HPMM datasets involved a total of 134,217,728 observation items for strong scaling and 524,288 observation items per core for weak scaling. In the HPMM case, the number of items in each segment followed a continuous uniform distribution for strong scaling and a Gaussian distribution for weak scaling. The grain size was 1024. Figures 13 and 14 show the results, indicating a near-linear speedup.

5 Conclusions

We have investigated a divide-and-conquer-based parallel computation strategy for machine-learning algorithms. Our approach not only reduces task-scheduling overheads dramatically, but also realizes efficient load balancing by cooperating with a work-stealing scheduler. Furthermore, the divide-and-conquer algorithm derives parameters without requiring mutual exclusion or atomic operations with shared variables by using a buffering solution that avoids the bottleneck of cache-coherence protocols in NUMA environments. We tested the scalability of our approach with both 80-core and 160-core NUMA computers and found that the divide-and-conquer solution achieved far superior scalability to FIFO-based parallelization and showed robustness against load-imbalanced datasets.

In this work, we have evaluated our method only for shared-memory computers with little discussion of reference locality because the buffering solution dealt with much of the reference-locality problem effectively. However, we intend to investigate ADCA for distributed-memory environments in future work, and the buffering solution alone would not be sufficient provision against the greater latency of message passing. Considering the memory access patterns of machine-learning algorithms, there is room for improved reference locality, given that the algorithms access observation items one by one continuously at each learning step, and repeat the steps many times. That is, after the scheduler assigns tasks and observation subsets to nodes before processing, enabling each task to access only its local data, the scheduler could improve reference locality by sending the same tasks to the same nodes at every step, as proposed by Yang et al. [26].

In other future work, our approach will seek to exploit the characteristics of GPUs. As stated in Sect. 2.1, GPUs are hardly applicable to graphical models on their own. Fortunately, a CPU can cooperate with a GPU by using CUDA

[57, 58], and a GPU could realize load balancing by cooperating with ADCA through CUDA. As shown in Algorithm 2, our approach employs a loop at the grain level. We expect GPUs to be able to accelerate this loop.

Acknowledgment. This work was supported by the CPS-IIP (<http://www.cps.nii.ac.jp>.) project under the research promotion program for national challenges *Research and development for the realization of the next-generation IT platforms* of the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan. The experimental environment was made available by Assistant Prof. Hajime Imura at the Meme Media Laboratory, Hokkaido University, and Yasuhiro Shirai at HP Japan Inc.

A General EM Algorithm

A.1 EM on GMM

The GMM is a popular probabilistic model described by a weighted linear sum of K normal distributions:

$$p(\mathbf{x}) = \sum_{k=1}^K w_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, S_k), \quad (1)$$

where w_k is the weight, $\boldsymbol{\mu}_k$ is the mean, and S_k is the covariance matrix of the k th normal distribution. An observation item \mathbf{x} is generated by a normal distribution selected with a probability of w_k . We transcribe parameters $\theta_k = (w_k, \boldsymbol{\mu}_k, S_k)$ for the sake of simplicity, and θ is the set of all θ_k . The likelihood function $\mathcal{L}(\theta)$ indicates how likely it is that the probabilistic model regenerates the training dataset. Assuming independence among observation items, $\mathcal{L}(\theta)$ is equal to the joint probability of all observation data. \mathcal{L} is defined in log-likelihood terms because $p(\mathbf{x}_n|\theta)$ is very small:

$$\mathcal{L}(\theta) = \sum_n \log \sum_k w_k \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, S_k). \quad (2)$$

In the EM context, we need only maximize \mathcal{L} . However, because a GMM is a latent-variable model, it requires step-by-step improvement. The posterior probability q_{nk} that the n th observation item \mathbf{x}_n is generated by the k th normal distribution is:

$$q_{nk} = \frac{w_k \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, S_k)}{\sum_k w_k \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, S_k)}. \quad (3)$$

Of the two repeated steps, the E-step calculates q_{nk} for all pairs of data \mathbf{x}_n and the k th normal distribution, and the M-step updates the parameters as follows:

$$\hat{w}_k = \frac{1}{N} \sum_n q_{nk}, \quad (4)$$

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{N\hat{w}_k} \sum_n^N q_{nk} \mathbf{x}_n, \quad (5)$$

$$\hat{S}_k = \frac{1}{N\hat{w}_k} \sum_n^N q_{nk} (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)^T (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k). \quad (6)$$

The E-step and M-step are repeated alternately until \mathcal{L} converges. In practice, the covariance matrix S_k is assumed to be a diagonal matrix and the calculation is therefore simplified as follows:

$$\hat{S}_{kd} = \frac{1}{N\hat{w}_k} \left(\sum_n^N \hat{q}_{nk} \mathbf{x}_{nd}^2 \right) - \hat{\boldsymbol{\mu}}_{kd}^2. \quad (7)$$

In the E-step, $N \times K$ q_{nk} is calculated, and in the M-step, q_{nk} is summed in the N axis and the parameter θ_k is updated. However, the posterior table can be too large and can exceed the hard-disk capacity when N is very large. Because of poor memory throughput, the processing speed will then degrade greatly. To avoid this condition, the parallel EM algorithm requires a large memory space.

A.2 EM on HPMM

Kinoshita et al. used an HPMM to detect traffic incidents [10]. They assumed that probe-car records follow a hierarchical PMM and that each road segment has its own local parameters. In their model, the probability of a single record \mathbf{x} in a segment s is described as follows:

$$p(\mathbf{x}|s) = \sum_{k=1}^K w_{sk} \mathcal{P}(\mathbf{x}; \boldsymbol{\mu}_k), \quad (8)$$

where w_{sk} is the k th Poisson distribution's weight in segment s , and $\boldsymbol{\mu}_k$ is the k th Poisson distribution's mean. w_{sk} is particular to the segment, whereas $\boldsymbol{\mu}_k$ is common to all segments. The log-likelihood $\mathcal{L}(\theta)$ is defined as follows:

$$\mathcal{L}(\theta) = \sum_{s=1}^S \sum_{n=1}^{N_s} \log \sum_{k=1}^K w_{sk} \mathcal{P}(\mathbf{x}_{sn}; \boldsymbol{\mu}_k), \quad (9)$$

where N_s is the number of records in segment s . As for GMMs, we must calculate the posterior probability q_{snk} that the n th record \mathbf{x}_{sn} in segment s is generated by the k th Poisson distribution for all pairs of (s, n, k) in each E-step:

$$q_{snk} = \frac{w_{sk} \mathcal{P}(\mathbf{x}_{sn}; \boldsymbol{\mu}_k)}{\sum_{k=1}^K w_{sk} \mathcal{P}(\mathbf{x}_{sn}; \boldsymbol{\mu}_k)}. \quad (10)$$

In the M-step, the weight w_{sk} and mean μ_k are recalculated:

$$\hat{w}_{sk} = \frac{1}{N_s} \sum_{n=1}^{N_s} q_{snk}, \quad (11)$$

$$\hat{\mu}_k = \frac{\sum_{s=1}^S \sum_{n=1}^{N_s} q_{snk} \mathbf{x}_{sn}}{\sum_{s=1}^S \sum_{n=1}^{N_s} q_{snk}}. \quad (12)$$

Each road segment has a massive number of records, with the actual number varying greatly from segment to segment. This implies that we should take measures against load imbalance.

References

1. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenkerand, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, June 2010
2. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., MacCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, April 2012
3. Power, R., Li, J.: Piccolo: building fast, distributed programs with partitioned tables. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, October 2010
4. Huang, C., Chen, Q., Wang, Z., Power, R., Ortiz, J., Li, J., Xiao, Z.: Spartan: a distributed array framework with smart tiling. In: Proceedings of the USENIX Annual Technical Conference, July 2015
5. Dijkstra, E.W.: Cooperating sequential processes. EWD: EWD123 (1968)
6. Mohr, E., Kranz Jr., D.A., Halstead, R.H.: Lazy task creation: a technique for increasing the granularity of parallel programs. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, May 1990
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, August 1995
8. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. *J. Roy. Stat. Soc. Ser. B (Methodol.)* **39**(1), 1–38 (1977)
9. McLachlan, G.J., Krishnan, T.: The EM Algorithm and Extensions. Wiley, Hoboken (2008)
10. Kinoshita, A., Takasu, A., Adachi, J.: Traffic incident detection using probabilistic topic model. In: Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference, March 2014

11. Kinoshita, A., Takasu, A., Adachi, J.: Real-time traffic incident detection using a probabilistic topic model. *Inf. Syst.* **54**(C), 169–188 (2015)
12. Pereira, S.S., Lopez-Valcarce, R., Pages-Zamora, A.: A diffusion-based EM algorithm for distributed estimation in unreliable sensor networks. *IEEE Signal Process. Lett.* **20**(6), 595–598 (2013)
13. Chen, J., Salim, M.B., Matsumoto, M.: A gaussian mixture model-based continuous boundary detection for 3d sensor networks. *Sensors* **10**(8), 7632–7650 (2010)
14. Miura, K., Noguchi, H., Kawaguchi, H., Yoshimoto, M.: A low memory bandwidth gaussian mixture model (GMM) processor for 20,000-word real-time speech recognition FPGA system. In: 2008 International Conference on ICECE Technology, December 2008
15. Gupta, K., Owens, J.D.: Three-layer optimizations for fast GMM computations on GPU-like parallel processors. In: IEEE Workshop on Automatic Speech Recognition & Understanding, December 2009
16. Stauffer, C., Grimson, W.E.L.: Adaptive background mixture models for real-time tracking. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, June 1999
17. Li, H., Achim, A., Bull, D.R.: GMM-based efficient foreground detection with adaptive region update. In: Proceedings of the 16th IEEE International Conference on Image Processing, November 2009
18. Patel, C.I., Patel, R.: Gaussian mixture model based moving object detection from video sequence. In: Proceedings of the International Conference and Workshop on Emerging Trends in Technology, February 2011
19. Song, Y., Li, X., Liu, Q.: Fast moving object detection using improved gaussian mixture models. In: International Conference on Audio, Language and Image Processing, July 2014
20. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. In: *Neurocomputing: Foundations of Research*, January 1988
21. Liu, Z., Li, H., Miao, G.: MapReduce-based backpropagation neural network over large scale mobile data. In: Sixth International Conference on Natural Computation, August 2010
22. Gu, R., Shen, F., Huang, Y.: A parallel computing platform for training large scale neural networks. In: IEEE International Conference on Big Data, October 2013
23. Hillis, W.D., Steele Jr., G.L.: Data parallel algorithms. *Commun. ACM Spec. Issue Parallelism* **29**(12), 1170–1183 (1986)
24. Flynn, M.J.: Some computer organizations and their effectiveness. *IEEE Trans. Comput.* **C-21**(9), 948–960 (1972)
25. Kwedlo, W.: A parallel EM algorithm for Gaussian mixture models implemented on a NUMA system using OpenMP. In: 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), February 2014
26. Yang, R., Xiong, T., Chen, T., Huang, Z., Feng, S.: DISTRIM: parallel GMM learning on multicore cluster. In: IEEE International Conference on Computer Science and Automation Engineering (CSAE), May 2012
27. Wolfe, J., Haghighi, A., Klein, D.: Fully distributed EM for very large datasets. In: Proceedings of the 25th International Conference on Machine Learning, July 2008
28. Kumar, N.S.L.P., Satoor, S., Buck, L.: Fast parallel expectation maximization for gaussian mixture models on GPUs using CUDA. In: 11th IEEE International Conference on High Performance Computing and Communications, June 2009

29. Machlica, L., Vanek, J., Zajic, Z.: Fast estimation of gaussian mixture model parameters on GPU using CUDA. In: 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), October 2011
30. Altinigneli, M.C., Plant, C., Bohm, C.: Massively parallel expectation maximization using graphics processing units. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 2013
31. Bergstrom, L., Reppy, J.: Nested data-parallelism on the GPU. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, September 2012
32. Lee, H., Brown, K.J., Sujeeth, A.K., Rompf, T., Olkotun, K.: Locality-aware mapping of nested parallel patterns on GPU. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, December 2014
33. Feeley, M.: A message passing implementation of lazy task creation. In: Halstead, R.H., Ito, T. (eds.) PSC 1992. LNCS, vol. 748, pp. 94–107. Springer, Heidelberg (1993). doi:[10.1007/BFb0018649](https://doi.org/10.1007/BFb0018649)
34. Umatani, S., Yasugi, M., Komiya, T., Yuasa, T.: Pursuing laziness for efficient implementation of modern multithreaded languages. In: Veidenbaum, A., Joe, K., Amano, H., Aiso, H. (eds.) ISHPC 2003. LNCS, vol. 2858, pp. 174–188. Springer, Heidelberg (2003). doi:[10.1007/978-3-540-39707-6_13](https://doi.org/10.1007/978-3-540-39707-6_13)
35. Acar, U.A., Chargueraud, A., Rainey, M.: Scheduling parallel programs by work stealing with private dequeues. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2013
36. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, May 1998
37. Min, S.J., Iancu, C., Yelick, K.: Hierarchical work stealing on manycore clusters. In: Fifth Conference on Partitioned Global Address Space Programming Models, October 2011
38. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, May 2011
39. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.* **26**(2), 110–124 (2012)
40. Nakashima, J., Nakatani, S., Taura, K.: Design and implementation of a customizable work stealing scheduler. In: 3rd International Workshop on Runtime and Operating Systems for Supercomputers, June 2013
41. Kranz, D.A., Halstead, R.H., Mohr Jr., E.: Mul-T: a high-performance parallel lisp. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, June 1989
42. Wheeler, K.B., Murphy, R.C., Thain, D.: Qthreads: an API for programming with millions of lightweight threads. In: IEEE International Symposium on Parallel and Distributed Processing, April 2008
43. Molka, D., Hackenberg, D., Shone, R., Muller, M.S.: Memory performance and cache coherency effects on an intel nahalem multiprocessor system. In: 18th International Conference on Parallel Architectures and Compilation Techniques, September 2009
44. Molka, D., Hackenberg, D., Schone, R., Nagel, W.E.: Cache coherence protocol and memory performance of the intel haswell-EP architecture. In: 44th International Conference on Parallel Processing, September 2015

45. Charles, P., Donawa, C., Ebcioğlu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 2005
46. Callahan, D., Chamberlain, B.L., Zima, H.P.: The cascade high productivity language. In: 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, April 2004
47. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, vol. 6, December 2004
48. Furmento, N., Goglin, B.: Enabling high-performance memory migration for multithreaded applications on Linux. In: IEEE International Symposium on Parallel & Distributed Processing, May 2009
49. Lameter, C.: NUMA (non-uniform memory access): an overview. *Queue* **11**(7), 40 (2013)
50. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.: GraphLab: a new framework for parallel machine learning. In: Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence, June 2010
51. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. In: Proceedings of the VLDB Endowment, April 2012
52. Hamidouche, K., Falcou, J., Etiemble, D.: A framework for an automatic hybrid MPI+ openMP code generation. In: Proceedings of the 19th High Performance Computing Symposia, April 2011
53. Si, M., Pena, A.J., Balaji, P., Takagi, M., Ishikawa, Y.: MT-MPI: multithreaded MPI for many-core environments. In: Proceedings of the 28th ACM International Conference on Supercomputing, June 2014
54. Luo, M., Lu, X., Hamidouche, K., Kandalla, K., Panda, D.K.: Initial study of multi-endpoint runtime for MPI+ openMP hybrid programming model on multi-core systems. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2014
55. Kawakatsu, T., Kinoshita, A., Takasu, A., Adachi, J.: Highly efficient parallel framework: a divide-and-conquer approach. In: Chen, Q., Hameurlain, A., Toumani, F., Wagner, R., Decker, H. (eds.) DEXA 2015. LNCS, vol. 9262, pp. 162–176. Springer, Heidelberg (2015). doi:[10.1007/978-3-319-22852-5_15](https://doi.org/10.1007/978-3-319-22852-5_15)
56. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, June 1998
57. Kirk, D.B., Hwu, W.W.: *Processors, Programming Massively Parallel: A Hands-on Approach*. Morgan Kaufmann, San Francisco (2010)
58. Nvidia. CUDA C programming guide version 6.5, August 2014

Transactions on Large-Scale Data- and
Knowledge-Centered Systems XXVIII
Special Issue on Database- and Expert-Systems
Applications

Hameurlain, A.; Küng, J.; Wagner, R.; Chen, Q. (Eds.)

2016, XI, 157 p. 43 illus., Softcover

ISBN: 978-3-662-53454-0