Best-Effort Parallel Execution Framework for Recognition and Mining Applications

Jiayuan Meng[†‡], Srimat Chakradhar[†], and Anand Raghunathan[†]
[†] NEC Laboratories America, Princeton, NJ
[‡] Department of Computer Science, University of Virginia, Charlottesville, VA

Abstract

Recognition and mining (RM) applications are an emerging class of computing workloads that will be commonly executed on future multi-core and many-core computing platforms. The explosive growth of input data and the use of more sophisticated algorithms in RM applications will ensure, for the foreseeable future, a significant gap between the computational needs of RM applications and the capabilities of rapidly evolving multi- or many-core platforms. To address this gap, we propose a new parallel programming model that inherently embodies the notion of best-effort computing, wherein the underlying parallel computing environment is not expected to be perfect. The proposed best-effort computing model leverages three key characteristics of RM applications: (1) the input data is noisy and it often contains significant redundancy, (2) computations performed on the input data are statistical in nature, and (3) some degree of imprecision in the output is acceptable. As a specific instance of a best-effort parallel programming model, we describe an "iterative-convergence" parallel template which is used by a significant class of RM applications. We show how the best-effort computing template can be used to not only reduce computational workload, but to also eliminate dependencies between computations and further increase parallelism. Our experiments on an 8-core machine demonstrate a speed-up of 3.5X and 4.3X for the K-means and GLVQ algorithms, respectively, over a conventional parallel implementation. We also show that there is almost no material impact on the accuracy of results obtained from best-effort implementations in the application context of image segmentation using K-means and eye detection in images using GLVQ.

1 Introduction

Recognition and *Mining* represent a significant class of emerging applications that will run on future multi-core and many-core computing platforms. They are expected to address the digital data explosion problem by enabling computers to model objects or events of interest to the user and use such models to search through massive amounts of data.

The paradigm shift to mainstream parallel processing is expected to enable applications to leverage device scaling and the resulting increase in chip densities in accordance with Moore's law, without concomitant increases in clock frequency. Recognition and Mining applications present abundant parallelism, and therefore stand to readily benefit from computing platforms with increasing numbers of cores.

RM applications are witnessing an explosive growth in input data, an increased use of sophisticated data processing algorithms, and a rising demand for real-time response. Therefore, for the foreseeable future, we expect a significant gap between the computational requirements of RM workloads, and the capabilities of emerging multi-core and many-core platforms. The success and adoption of recognition and mining applications will depend on technologies that effectively bridge this "computation gap".

RM applications share several unique characteristics: they accept input data that is noisy and redundant, they perform computations that are statistical in nature, and they can produce a large number of seemingly different solutions that are all considered acceptable (we refer to these characteristics as the "forgiving nature" of RM applications). We exploit this forgiving nature by proposing a parallel programming model for RM applications that inherently embodies the notion of "best-effort computing", wherein the computations presented to the platform are executed on a best-effort basis, *i.e.*, they are not always guaranteed to be executed. This is inspired by the notion of best-effort packet delivery in the Internet Protocol (IP) — in spite of debates about its benefits [6], we believe that it is one of the fundamental factors that have enabled the Internet to rapidly scale to meet the explosion in the volume of traffic.

We explore best-effort parallel computing in the context of a domain-specific parallel template for "iterative convergence" algorithms, which represent a significant class of RM algorithms. We demonstrate that iterative convergence algorithms possess several interesting properties that can be leveraged for the purpose of best-effort computing, and present various best-effort strategies. These strategies can be used for classification of computations into two cate-

gories: optional computations that may be dropped if necessary, and mandatory computations that must be completed in order to maintain integrity of the output results. Moreover, some strategies may eliminate dependencies between computations and further increase parallelism.

We apply the best-effort (BE) model to two important RM applications: K-means (an unsupervised clustering technique), and GLVQ (a supervised, classification technique). Both these applications employ algorithms that are iterative and converging. Therefore, we use the iterative-convergence template to express the two algorithms. For K-means application, we improve performance by drastically reducing the raw computation workload. For GLVQ application, we improve performance by eliminating potential task dependencies. Reduction in dependencies leads to more parallel tasks, thereby improving performance. Our experiments on a 2-way, quad-core Xeon show that K-means application with BE model can be accelerated by a factor of 3.5X as compared to a traditional parallel implementation of K-means on the 2-way, quad-core Xeon. For the GLVQ application with BE model, we obtained a speed up of 4.3X as compared to a traditional parallel implementation.

2 Best-effort Computing Model

To bridge the gap between RM applications' demand for performance and the capabilities of future computing platforms, we propose a radically new parallel programming model, called the Best-effort (BE) model. Our BE model, which is illustrated in Figure 1, serves as a run-time environment that introduces unreliability in order to accelerate RM applications by exploiting the forgiving nature that is inherent in these applications.

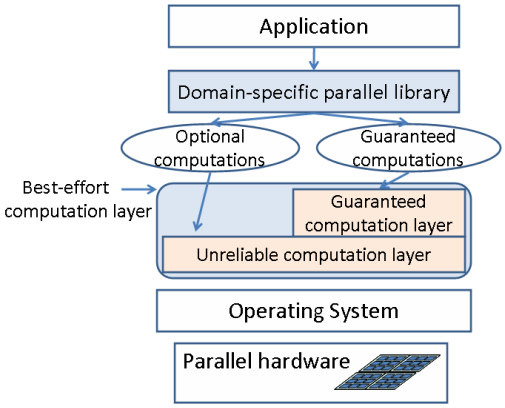


Figure 1. Best-effort computing model overview

We first make a fundamental change in the contract between applications and the computing environment (parallel hardware, OS and run-time libraries): the computing environment is unreliable and it may drop (i.e. not execute) some of the computations requested by the application (every application can be viewed as a collection of smaller computations).

This view is similar to the Internet Protocol model in computer networking where packets may be dropped by the network. IP protocol has been successful in managing ever-increasing volume of packet traffic for over three decades by simply reserving the right to drop packets, if necessary. By sacrificing reliability, it has become possible to build simpler and faster networks. A similar strategy can be used to build simpler and faster computing systems by reserving the right to drop computations due to a variety of reasons: defects in hardware, real-time constraints on response times, excessive computation load, power constraints or, as we show in this paper, to accelerate applications.

Unreliability of the underlying computing environment forces the application to re-structure its workload into optional and mandatory computations. This is again similar to re-structuring of network applications today to utilize the unreliable UDP protocol or the reliable TCP protocol, both of which are realized on the unreliable IP protocol. Our BE model provides high-level programming templates to easily and intuitively express various recognition and mining algorithms for execution on a parallel, unreliable computing environment. Using the BE model, applications can easily specify optional and mandatory computations. In this paper, we show how applications can leverage best-effort computing in two different ways: (1) drop computations to reduce overall workload and improve performance, or (2) relax dependencies between tasks, leading to higher performance through more task parallelism. Our BE model allows the application to easily experiment with a variety of dropping criteria for optional computations. The BE model run-time implements the best-effort strategies, computation dropping criteria and manages the execution of computations that cannot be dropped. Like TCP protocol in computer networking, the BE model run-time also implements a mechanism to ensure reliable execution of a mandatory computation by repeated re-scheduling of the mandatory computation when necessary.

In this paper, we explore best-effort computing in the specific context parallel implementation of Recognition and Mining algorithms on contemporary multi-core platforms. The underlying computing platforms (hardware and OS) that we consider are reliable, therefore we proactively drop optional computations in the best-effort layer in order to reduce computational workload and improve algorithms' parallel scalability. Guaranteed computations are passed through the best-effort layer onto the underlying computation platform, which executes them without any need for re-scheduling. The other facets of best-effort computing are beyond the scope of this paper, and we expect to explore them in our future work.

3 Motivation

For illustration purposes, we demonstrate the potential for best-effort computation using two commonly used algorithms from the RM domain: K-means and GLVQ. K-means is a widely used clustering algorithm and is also often used for unsupervised learning [17]. Generalized Learning Vector Quantization(GLVQ) [21, 24, 23] is a classification algorithm used in supervised learning, where the underlying data structure (a set of reference vectors) is updated as labeled training vectors are being processed. Although we only consider these two algorithms in our motivation and subsequent illustration, several other algorithms, including Fuzzy K-means [4], Support Vector Machines [8], and Principal Component Analysis [22], exhibit similar structure to K-means and GLVQ in that parallel computations are repeatedly performed to update values of specific data structures until a pre-specified convergence criteria is satisfied.

We demonstrate how best-effort computing can be applied to leverage the forgiving nature of RM algorithms in two different ways: reducing the amount of computation, and increasing parallelism. In both cases, performance can be significantly improved with only a small penalty in the quality of the result.

3.1 K-means: Potential for Computation Reduction

The K-means algorithm clusters a given set of points in a multi-dimensional space [17]. It begins by randomly picking several input points as cluster centroids. These cluster centroids are then refined in iterations until an iteration no longer changes any point's cluster assignment.

Each iteration performs three steps:

1. Compute the distance between every point and every cluster centroid.

2. Assign each point to the cluster centroid that the point is closest to. Points assigned to the same cluster centroid form a single cluster.

3. Re-compute the new centroid for each cluster to be the mean of all points in the cluster.

A common application of K-means clustering is to segment images into regions with similar color and texture characteristics. Image segmentation can be used as a preprocessing step for image content analysis or compression. We applied K-means to perform image segmentation by clustering all the pixels from a 1792×1616 image that represents a histological micrograph of tissue used for cancer diagnosis. A pixel in the RGB color space of the image corresponds to a point in the K-means clustering algorithm. Several characteristics warrant the use of best-effort computing. Figure 2(a) plots the number of points that change their memberships in each iteration. The figure shows that less than 1% of points change their memberships after around 20% of the iterations. Consider a point p whose membership has stabilized in iteration i, *i.e.*, it does not change clusters in subsequent iterations. All distance computations involving point p in iterations $i + 1$ and later will not have any impact on the final result. This indicates that future iterations could remove membership computation for points that have already "stablized". In practice, it is difficult to identify points that are *guaranteed* to not change clusters (due to a gradual change in cluster centroids, a point may not change clusters for several iterations but may eventually be assigned to a different cluster). However, as shown in our experiments, it is possible to identify points that will be *highly unlikely* to change clusters, and the associated computations (distance computations for these points) are likely to have a minimal impact on the final result. From a different perspective, Figure 2(b) indicates that cluster centroids migrate drastically during the first several iterations. This implies that these iterations do not demand very high accuracy in centroid computation. Therefore, it is possible that not all points have to be considered in the early iterations.

3.2 GLVQ: Potential for Dependency Reduction

GLVQ (*Generalized Learning Vector Quantization*) is a supervised learning algorithm used for classification [21]. During classification, the algorithm calculates the distance between the input vector and all pre-specified reference vectors (the training phase of the GLVQ algorithm creates reference vectors for each class). The input vector is assigned to the class with the nearest reference vector.

We focus on the computation-intensive training phase of the GLVQ algorithm. During this phase, the algorithm processes one training vector at a time. The algorithm performs the following three steps for each training vector:

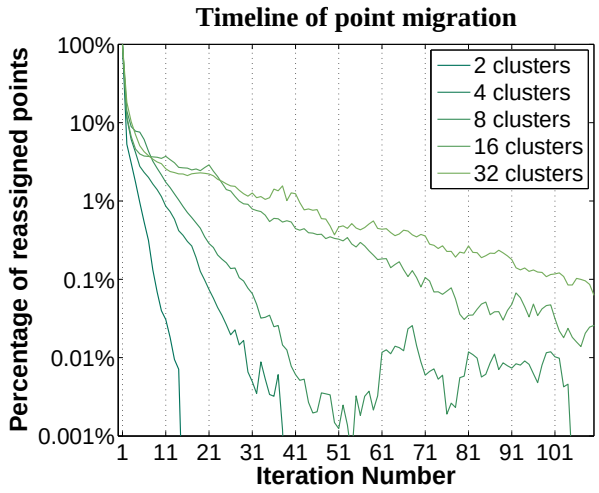
1. Compute distances between the training vector and all reference vectors.

2. Identify two reference vectors: (a) the closest reference vector R1 in the same labeled class as the training vector, and (b) the closest reference vector R2 that is not in the same labeled class as the training vector.

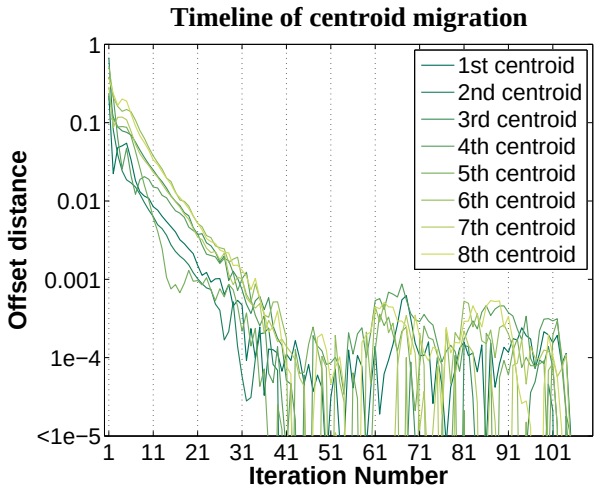
3. Suitably update the two reference vectors so that the training vector is closer to R1 and farther from R2.

This process is continued for all training vectors.

The training vectors are processed sequentially because of potential read-after-write (RAW) dependencies: reference vectors updated by the previous training vector may be used to calculate distances from the next training vector. However, most of the distance values are used only in Step 2 to select two reference vectors. Only two of the reference vectors will participate in Step 3, others are discarded.



(a) Percentage of points changing their membership in the timeline of iterations. Data is measured for K-means when it groups 2895872 points into 2, 4, 8, 16, and 32 clusters.



(b) The distances that each centroid migrates in the timeline of iterations. Data is measured for K-means when it groups 2895872 points into 8 clusters. A z-score transformation is performed prior to clustering to transform input dataset to have zero mean and unit variance.

Figure 2. K-means characterization

Therefore, in the case that two consecutive training vectors update different reference vectors, this inter-iteration RAW dependency becomes redundant.

To illustrate the potential for best-effort computing in the GLVQ training algorithm, we characterize the dependencies between computations in consecutive iterations (where each iteration corresponds to processing one training vector). We classify inter-iteration dependencies into true dependencies, where ignoring the dependency would have an impact on the result of the next iteration, and false dependencies, where ignoring the dependency would have no impact on the result of the next iteration. Note that a true dependency occurs only when two consecutive iterations update the same reference vector. We study the probability of true inter-iteration dependencies, which could also be viewed as the probability that consecutive iterations in parallel (ignoring the dependencies) would impact the result. The characterization is performed when GLVQ is used in the context of eye detection in images, which is an important step in face recognition. 300 images of eyes and 3000 images of non-eyes are used as input data. Images are generated from the Yale Face Database [13]. For each input image, histograms of gradients are extracted as training vectors, each has 512 dimensions. Two class, eye and non-eye, are present, each has 64 reference vectors. Due to the relative small number of eye images, they are replicated for 8 times each, yielding a total of 2400 eye images. A total of 5400 training vectors are fed to the training process in a randomized order.

The probability of incorrect relaxation of dependency (we refer to as false dependency relaxation) is shown in Figure 3. A trace is generated from a single threaded GLVQ algorithm. It records the reference vectors updated by each training vector. We then assume a parallel execution where training vectors are grouped into N batches, each is assigned to one of the N simultaneous threads. With the assumption that each training vector is processed in the same amount of time, we record the number of times that a reference vector is simultaneously selected by multiple training vectors for update. We show that the probability of false dependency relaxation stays below 30% even with 10 threads. Moreover, reference vectors impacted due to this incorrect relaxation may be rectified by following training vectors. The above observation indicates that we may be able to parallelize across training vectors to gain performance, while maintaining a reasonable modeling accuracy.

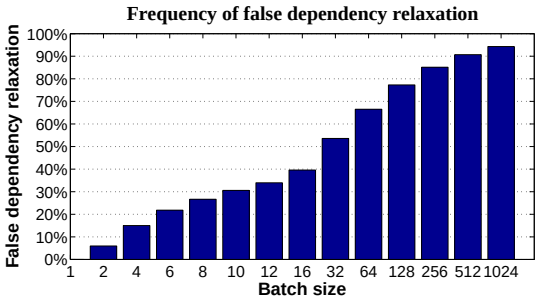


Figure 3. Percentage of false dependency relaxation vs. Number of simultaneously processed training vectors. Measurement is performed for GLVQ when it is learning from 5400 training vectors.

4 Best-Effort Computing for Iterative Convergence Paradigm

Many algorithms in the RM domain fit the so-called *iterative convergence* paradigm, wherein, a parallel computation is performed in an iterative manner until a convergence or termination condition is satisfied. As explained in the previous section, many characteristics of iterative convergence algorithms in the RM domain make them naturally suited to best-effort computing.

In this section, we first present a parallel programming template for iterative convergence algorithms that naturally embodies the concept of best-effort computing. This template provides a simple framework in which we can explore various tradeoffs involved in best-effort computing without unduly burdening the application programmer. We then discuss several best-effort strategies that can be used to exploit application properties in order to identify specific computations that can be dropped in order to mitigate the workload and data dependencies that can be ignored in order to increase parallelism. These strategies are implemented as libraries that allow application programmers to easily experiment with them.

4.1 An Abstract Programming Template

An abstract programming template for iterative-convergence algorithms that inherently embodies the best-effort computing model is shown in Figure 4. The `iterate{...}until converged(...)` construct iteratively performs a computation until the specified convergence criterion is satisfied. The convergence criterion may be specified as a test that depends on the values of data structures involved in the computation. For example, in the K-means algorithm, the classical convergence criterion is to check whether the data structure that stores the memberships of all points is unchanged since the previous iteration. Programmers have to specify the computation within each iteration as well as the convergence criteria.

```
iterate
{
  mask[1:M] = filter(...);
  parallel_iterate (i = 1 to M with mask[1:M] batch P)
  {
     ...
  }
} until converged(...);
```

Figure 4. Pseudocode of the best-effort iterative-convergence template.

The body of an iteration can be specified using a `parallel_iterate` construct, which differs from conventional parallel loop constructs (such as `parallel_for`) in two ways. First, a `mask` is used to indicate which iterations of the loop are optional computations and can hence be dropped. The `mask` is a Boolean array with entries that directly correspond to the iterations of the `parallel_iterate` operator. Second, the `batch` operator is used to specify that although the loop iterations may carry data dependencies, a certain number of iterations may be executed in parallel by ignoring these dependencies. The `mask` is produced by a `filter` operator which uses a filtering criterion that may depend on the values of data structures used in the computation. In the K-means algorithm, for example, the filtering criterion may be based on the number of iterations [1] for which a point has remained in the same cluster.

We observe that the proposed iterative convergence template can be used to explore best-effort computing in three different ways.

- The selection of appropriate filtering criteria that reduce the computations performed in each iteration.
- The selection of convergence criteria that decide when the iterations can be terminated.
- The use of the `batch` operator to relax data dependencies in the body of the `parallel_iterate`.

It is important to note that although the intent of best-effort computing is to improve performance, the filtering and convergence criteria themselves need to be computed. It is important to ensure that the additional computation introduced by these criteria does not outweigh the benefits derived therefrom. It is also important to ensure that the criteria chosen are amenable to highly parallel implementation, *i.e.*, do not introduce any bottlenecks to parallelism.

In summary, the best-effort template allows programmers to specify applications in a simple and intuitive manner. We show in Section 5 how different algorithms can be mapped to the proposed iterative convergence template. We next describe how various best-effort strategies can be explored in the context of the proposed iterative convergence template.

4.2 Best-Effort Strategies

We propose several best-effort strategies. All our strategies leverage unique characteristics of iterative-convergence based RM applications. Section 5.3 shows how these strategies can be embedded in iterative convergence templates for two example algorithms: K-means and GLVQ. Section 6 quantifies the benefits of these strategies on RM applications. These strategies are listed as follows:

- *Convergence-based Pruning*: Use converging data structures to speculatively identify computations that

[1] In the rest of the paper, we call the computation enclosed by the `iterate` operator as an *iteration*.

have minimal impact on results and eliminate them. In the K-means algorithm, two data structures are identified as converging: the membership array and the cluster centroids. Accordingly, two policies are proposed. Using the Iterative-Convergence programming template, the `filter` operator can mask off the computation for a point's membership if the membership barely changes in the past iterations. We name this strategy *conv.point*. An alternative strategy, *conv.center*, masks off the same computation if a point is assigned to a cluster with a centroid that hardly migrates.

- *Staged Computation*: Organize computation to proceed in stages. We consider fewer points in early stages and this can result in low accuracy of initial estimates of the results. However, we gradually consider more and more points in subsequent stages so that the initial estimates are refined into more accurate final results. Our *staged* computation strategy attempts to expedite overall convergence rate by consider fewer points in early stages. However, there is also a risk that we slow down the convergence rate if we do not select representative subset of points in the initial stages. For the K-means algorithm, we employ the staged computation strategy. The `filter` operator chooses only a subset of points in the beginning. Points are gradually added to the subset to refine the result.

- *Early Termination*: Aggregate statistics to estimate accuracy and terminate before absolute convergence. Fewer iterations are computed at the expense of partial convergence. The termination criteria is encoded by the `converged` operator in the programming template. This so-called *terminate* strategy can be easily adopted by K-means: the algorithm terminates when the percentage of unstable points — points that changed their memberships in the last iteration — drops below a given threshold.

- *Sampling*: Select a random subset of input data and use it to compute the results. This *sample* strategy is useful when significant redundancy is expected in the input data. Otherwise, a great portion of input information may be lost during computation, and accuracy may degrade drastically. In the Iterative-Convergence template, *Sampling* can be applied to K-means by enforcing the `filter` operator to choose the same subset of input data in all iterations when the cluster centroids are computed. The centroids are eventually used to compute memberships for all the points.

- *Dependency Relaxation*: Ignore potentially redundant dependencies across iterations. Iterations can then be parallelized, leading to more degree of parallelism or coarser parallel granularity (i.e., threads have larger tasks or a larger number of tasks). This coarser granularity of parallelism obscures the overhead in task and thread management. The GLVQ example uses the `batch` operator to exploit this extra granularity of parallelism. However, modeling accuracy may be impaired due to false dependency relaxations.

The above best-effort strategies can be used in combinations. For instance, *Sampling* can select a subset of input data before any other strategies are applied. In addition, *Early Termination* can be mixed with *Staged Computation* as a relaxed criteria that determines when to advance to the next stage. Furthermore, *Convergence-based Pruning* can be added to reduce the amount of computation in each stage.

Alternatively, some best-effort strategies can be made probabilistic. Consider *Convergence-based Pruning* when it is applied to K-means. If the membership is speculated as converged, its computation can be skipped only occasionally, given a pre-determined probability.

We show in Section 5 that the optimal strategy may vary depending on characteristics of the algorithm as well as the input data. Nevertheless, our parallel programming template allows easy evaluation of different strategies.

5 Case Study: K-means clustering

In this section, we discuss the application of the best-effort computing model, specifically, the best-effort iterative convergence parallel template, to the K-means clustering algorithm. We also evaluate the performance and quality-of-results for a best-effort parallel implementation of K-means in the context of an image segmentation application. In the following subsections, we show how the K-means algorithm is mapped to the best-effort iterative convergence template, describe the specific best-effort strategies that we evaluated, and finally the data sets used and experimental results obtained.

5.1 K-means using the best-effort iterative convergence template

The pseudo-code presented in Figure 5 shows how the k-means clustering algorithm can be expressed using the proposed best-effort iterative convergence template.

Recall that the K-means algorithm addresses the problem of clustering n points into K clusters. The data structures used in the algorithm are arrays that store the points, cluster centroids, distances of each point to the cluster centroids, and the cluster memberships. Initially, K random points are chosen as centroids. Each cluster has exactly one centroid. The function `random_select()` selects K random points. Then, depending on the specific `filter_strategy` that is used for filtering, the `filter()` function generates a mask array such that `mask[i] = 1` means that the i^{th} point will be considered for further computation during the current iteration. In other words, the computations involving point i are guaranteed computations. Computations involving points

```
/* N points, K clusters (or centroids), each point has D dimensions */
float points[1:N], centroids[1:K]
int memberships[1:N]; /*cluster membership for a point */
int distances [1:K]; /* distance of point from K centroids */

Kmeans_best_effort ( ) {
  /* randomly choose K points as centroids */
  centroids[1:K] = random_select(points, K);
  iterate
  {
    /*generate mask*/
    Mask mask[1:N] = filter (memberships[1:N], filter_strategy);

    /* compute only for un-masked points */
    parallel_iterate (i = 1 to N with mask[1:N] batch N)
    {
      distances[1:K] =  compute_distances (points[i], centroids[1:K]);
      memberships[i] = argmin (distances[1:K]);
    }

    /* update cluster centroids: parallel reduction */
    centroids[1:K] = compute_means (points[1:N], memberships[1:N]);

  /* terminate if all elements of memberships array are unchanged */
  } until converged(memberships, unchanged);
}
```

Figure 5. Pseudocode of K-means in the best-effort iterative-convergence template.

whose entries in the `mask[]` array is 0 are optional computations. The `parallel_iterate` loop only processes points whose mask value is 1. Note that in the case of K-means, the `batch` operator is given the parameter N, since all iterations of this loop are independent. The function `compute_distances()` computes the distance of the i^{th} point from all the K centroids. The function `argmin()` computes the index of the centroid that is closest to the i^{th} point. The i^{th} point is then assigned to the cluster corresponding to the closest centroid. Then, the `compute_means()` function computes the new centroid for all points in a cluster. Finally, depending on the specific best-effort convergence criteria in place, the program decides when to terminate the K-means algorithm. The original K-means convergence criterion, which is illustrated in the pseudo-code, is to check whether the values of the `memberships[]` array are unchanged since the previous iteration.

5.2 Best-effort strategies for K-means

We evaluated several different best-effort strategies for the K-means algorithm, including five different best-effort filtering criteria, and one best-effort convergence criterion, which are described below. It is important to note that all the filtering criteria are fully parallel, *i.e.*, they can be evaluated independently on each point. This is especially important for parallel implementation since we would like to avoid a situation where the computations added for the purpose of best-effort become parallel execution bottlenecks. The convergence criterion is evaluated as a parallel reduce operation, similar to the original convergence criterion of the K-means algorithm.

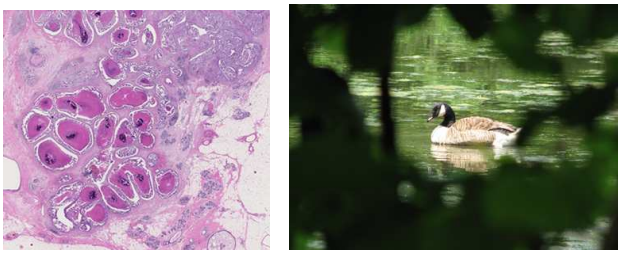
The strategies presented below are parameterized - we have developed a generic run-time library that implements these strategies so that the application programmer only needs to instantiate the appropriate strategy and choose values of the parameters. This makes it easy for the programmer to experiment with different strategies and parameter values.

- *terminate*: The convergence criterion is made to return $True$ when less than $T\%$ of the points have changed their memberships since the last iteration.
- *sample*: $P\%$ of the n points are randomly sampled. For all sampled points, corresponding mask bits in the `mask[]` array are set to 1 so that these points participate in computations during the current iteration.
- *stage*: The filtering criterion starts with a subset of points and gradually adds points in stages. Initially, the `mask[]` array is set to 1 for only $\frac{1}{256}$ of the n points. The number of points considered grows geometrically for subsequent stages. A total of S stages are deployed, where the last stage considers all the points. The algorithm advances to the next stage when the convergence criterion evaluates to $True$ in the previous stage.
- *conv.point*: The filtering criterion identifies points whose memberships have remained the same in the last N iterations. The `mask[]` array entries are set to 0 for these points that are deemed to have "converged", and to 1 for all other points.
- *conv.center*: The filtering criterion identifies points whose corresponding centroids have changed by a distance of greater than D since the previous iteration. The `mask[]` array entries are set to 1 for such points, and 0 for all other points. Note that in all our experiments with K-means, the input dataset is transformed to have zero mean and unit variance using the z-score transformation [18]. The distance measurement is based upon this normalized space.
- *conv.vacation*: A heuristics measure is used to determine for each point, how many iterations of membership computation can be bypassed before recomputing it again. The number of bypassed iterations is named as *vacation length (β)*. The heuristic is modeled as a function of the *distinguishing degree (α)* that quantifies how distinquishing the point's current cluster membership is, and the likelihood that it is going to be affected by an updated centroid. It is defined as $\alpha = \frac{D_s-(D_m+\delta)}{D_s+(D_m+\delta)}$, where D_m is the distance of the point to its nearest centroid, D_s is the disntance to its second nearest centroid, and δ is the offset distance of

the centroid that moved the most in the previous iteration. Based on extensive experiments with K-means when applied to image segmentation, we found that selecting $\beta = \max(13.5 \times \alpha - 2.7, 0)$ provided a good tradeoff between performance and accuracy.

5.3 Results

We implement the best-effort K-means algorithm using Intel's TBB [11] and evaluated it on a Dell Poweredge 2950 8-core machine (2-way SMP system with Intel Xeon E5320 quad-core CPUs) and 8GB memory running RedHat Enterprise Linux 5. We use two different datasets to evaluate the parallel best-effort K-means implementation: (i) images from histological micrographs of tissue samples used for cancer diagnosis (Figure 5.3(a)), and (ii) a scenic image that has a small depth of field, including a large blurred area of a similar color while the focused details are limited to a small area (Figure 5.3(b)).



(a) *Cancer*: A histological tissue micrograph for cancer diagnosis (1600×1200) (b) *Goose*: A Canada goose behind a blurry foreground (1792×1616)

Figure 6. Images used for evaluating best-effort K-means

We evaluate the effectiveness of each best-effort strategy with respect to performance and error introduced in the result due to the use of best-effort computing. In order to establish a performance baseline and golden result in order to determine the error, we execute the K-means algorithm without any best-effort strategy, *i.e.*, the filtering operation is not performed, and the iterations proceed until no point changes its membership. This baseline result is then compared with those resulting from best-effort strategies. The percentage of points that end up with different cluster memberships is reported as the error rate. In all experiments, we use K-means to generate 8 clusters, *i.e.*, K=8.

Individual best-effort strategies: Figure 7 presents error *vs.* performance curves resulting from each of the best-effort strategies described in the previous sub-section. For each strategy, we vary the parameter(s) involved in order to generate implementations with varying error *vs.* performance tradeoffs. The baseline is shown in the *terminate* strategy with $T\%$ set to 0%, i.e., no best-effort strategies are applied. All best-effort strategies reduce the execution

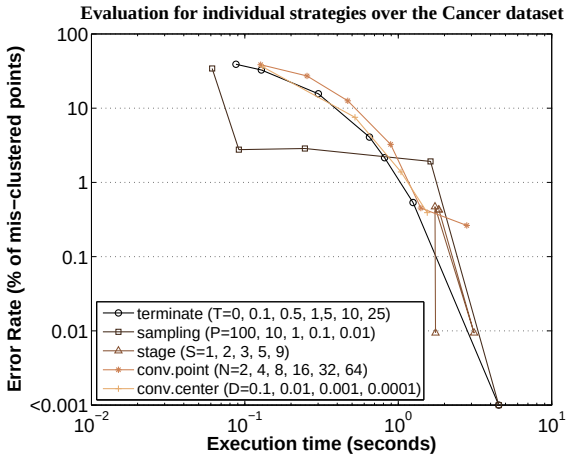


Figure 7. Error vs. performance for individual best-effort strategies (K-means based segmentation of histological micrograph image).

time at the expense of minimal loss in accuracy. With an error rate of less than 1%, execution time can be reduced from 4.5 seconds to 1.3 seconds — a 3.5X speedup. The *sample* strategy can further reduce the execution time to 0.1 seconds at an error rate of 3%, while other strategies can reduce the execution time to 0.6 seconds with an error rate of 4%. The *stage* strategy results in the best accuracy. With 9 stages, it results in an execution time of 1.75 seconds while the error rate is maintained as low as 0.01%. Meanwhile, all best-effort strategies achieve linear performance scaling when we increase the number of threads from one to eight. This indicates that both the original K-means algorithm, and the computations added for implementing the best-effort strategies (filter and convergence criteria) are efficiently parallelized.

Combined best-effort strategies: We investigate the space of combined best-effort strategies, as mentioned in Section 4.2. In most circumstances, combined strategies have more performance gains at a lower error rate compared to individual strategies. The error and performance for several combined policies are plotted in Figure 8(a). The *terminate* strategy is included in all the combinations and the termination criteria($T\%$) is varied among 0%, 0.1%, 0.5%, 1%, 5%, 10%, and 25% to generate the error *vs.* performance curves. The parameter values for other strategies are listed in the legend. In both Figure 8(a) and Figure 8(b), we include the curve of the individual *terminate* strategy for comparison, and the baseline is also shown in this curve with $T\%$ set to 0%. We show that in most cases, the *sample* strategy is very effective in trading off error *vs.* performance. However, it can be further improved by combining it with the *stage* and *conv.center* policies.

In order to illustrate that the optimal strategies and pa-

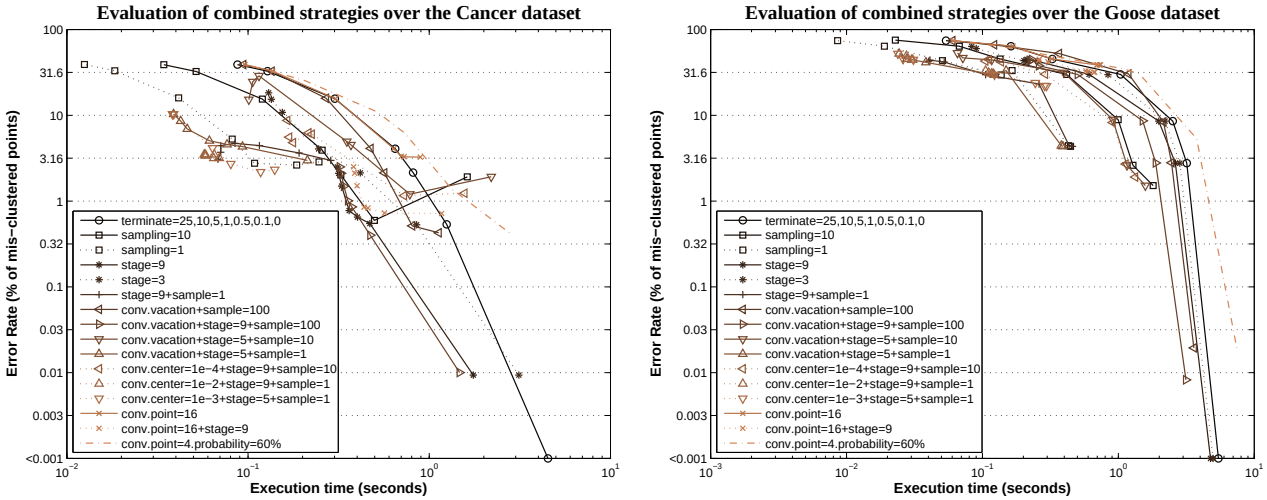


Figure 8. Error vs. Execution time for combined best-effort strategies when applied to K-means based image segmentation for (a) Cancer and (b) Goose.

rameter values may vary depending on the algorithm and the input data, we repeat the experiments on combined best-effort strategies with a scenic image (Figure 5.3(b)) that has a small depth of field, introducing a large blurred area in a similar color while the focused details are limited to a small area. The results are presented in Figure 8(b). Although the *sample* strategy is very effective in some cases, the optimal sampling rate depends on the redundancy of the input data. While a sampling rate of 1% yields an error rate of 3% for the histological image, its error rate increases to more than 30% for the scenic image. On the other hand, the *stage* strategy demonstrates more robust results when combined with the *terminate* strategy with a termination parameter (T%) of 0.1% to 1%.

In summary, our experiments with best-effort computing applied to the K-means algorithm demonstrate that (i) significant performance improvements can be obtained due to reductions in computational workload with very small impact on quality of the result, and (ii) the computations introduced by the various best-effort strategies are highly parallel and do not impact the excellent parallel scaling behavior of the K-means algorithm.

6 Case Study: GLVQ training

In this section, we discuss the application of the best-effort iterative convergence template to the GLVQ training algorithm, and evaluate the performance *vs.* error tradeoffs achieved in the context of using GLVQ for eye detection from images. We demonstrate that best-effort computing results in a signifcant improvement in parallel scalability of GLVQ training.

6.1 GLVQ training using the best-effort iterative convergence template

The pseudo-code presented in Figure 9 shows how the GLVQ training algorithm can be expressed using the proposed best-effort iterative convergence template.

The GLVQ training algorithm processes N training vectors that are provided with corresponding labels that denote the class that each belongs to. A set of M reference vectors is created for each of the C classes. Since the algorithm only makes a single pass through the training vectors, the convergence criterion for the `iterate` operator is set to `TRUE`. The `parallel_iterate` operator iterates through the training vectors, and processes them in parallel batches of P vectors, relaxing any data dependencies between the P vectors that are in the same batch. In each iteration, the following computations are performed. The function `euclid()` computes the Euclidean distances from the i^{th} training vector to all reference vectors in all classes. Based on these distances, the function `choose_nearest_vectors()` selects two reference vectors — one from the correct class that is closest to the current training vector, and one from among all the incorrect classes that is closes to the current training vector. The function `update_refs()` updates these two reference vectors so that the former is pulled closer to the current training vector while the later is pushed away from it. The algorithm terminates when all the training vectors have been processed.

6.2 Best-effort strategy for GLVQ

GLVQ is conventionally parallelized by using multiple threads to perform distance computation from the training

```
/* C classes, M reference vectors per class, N labeled training vectors */
int C, M, N;
array ref_vecs [1:C][1:M]; /* reference vectors */
array train_vecs [1:N]; /* training vectors */
int labels[1:N];  /* class labels for each training vector */

int P; /* number of training vectors to process in parallel */

GLVQ_train_best_effort()
{
 iterate
 {
    /* sets all mask entries to 1 */
    Mask mask[1:N] = filter(NONE);
    parallel_iterate(i = 1 to N with mask [1:N] batch P)
    {
       /* calculate distance of the training vector to all the reference vectors */
       distances[1:C][1:M] = euclid(ref_vecs[1:C][1:M],train_vecs[i]);

       /* pick two reference vectors, target_ref is the nearest
        * one in the labeled class. other_ref is the nearest one in all other classes
        */
       target_ref, other_ref = choose_nearest_vectors(label[i],
          ref_vecs[1:C][1:M], distances[1:C][1:M]);

       /* update the two picked reference vectors */
       update_refs(target_ref, other_ref, distances[1:C][1:M]);
    }
 } until converged (TRUE);
}
```

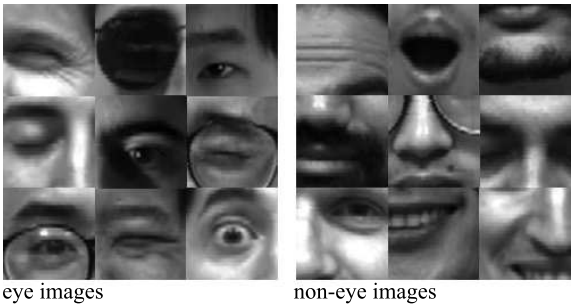
Figure 9. Pseudocode of GLVQ in the best-effort iterative-convergence template.

vector to all the reference vectors and finding the two closest reference vectors. However, the issue with this approach is that in many application scenarios a limited number of reference vectors encumbers effective parallelism. In our example of eye detection, a total of 128 reference vectors are present. With 8 threads, each thread calculates distances to only 16 reference vectors. As a result, the benefit of parallel execution is significantly reduced due to the overheads of parallel task creation and management.

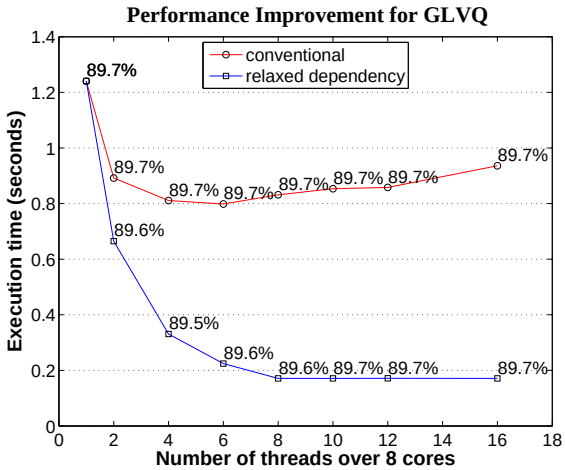
The use of the `parallel_iterate` operator exploits more parallelism and enforces a larger parallel granularity by relaxing dependencies between iterations that process consecutive training vectors. However, we risk the loss of accuracy due to the relaxation of RAW data dependencies that may be occasionally present. In our experiments, we find that the loss in accuracy that results from relaxing data dependencies is very small and hence acceptable. Moreover, it is also possible to detect data dependencies (multiple iterations in a parallel batch update the same reference vectors), and pre-execute the dependent iterations sequentially.

6.3 Results

We implement the best-effort GLVQ training algorithm using Intel's TBB [11] and evaluated it on a Dell Poweredge 2950 8-core machine (2-way SMP system with Intel Xeon E5320 quad-core CPUs) and 8GB memory running RedHat Enterprise Linux 5. GLVQ is trained to detect eyes and label images accordingly. During the training phase, a model is built after a large set of labeled eye and non-eye images, as shown in Figure 10(a). In our experiments, the training data set consisted 2400 eye images and 3000 non-eye images. The training process is accelerated using the best-effort strategy of relaxing dependencies across iterations. The model that is produced by the training process is then used to classify a different test set of images in order to evaluate its accuracy.



(a) Sample input images for GLVQ training.



(b) Execution time vs. number of threads for conventional and best-effort implementations of GLVQ training. Annotations to each point denote the corresponding classification accuracy.

Figure 10. Evaluation of GLVQ training for eye detection

Figure 10(b) compares the conventional and best-effort implementations of GLVQ training for eye detection in images. We observe that the conventional parallel GLVQ implementation yields a peak speedup of 1.7X with 6 threads (larger numbers of threads result in a degradation in performance). The limited speedup is resulted from the small amount of workload in each parallel thread, which leads to frequent thread management with significant overhead. The best-effort GLVQ implementation, on the other hand,

obtains a coarser granularity of parallelism and generates threads with heavier workload that last longer, reducing the thread management and task distribution overhead. Consequently, it yields a peak speedup of 7.3X and performance continues to scale with 8 threads.

We test the classification accuracy by using the trained reference vectors to classify 115 eye images and 1350 non-eye images. The model generated from the conventional algorithm correctly classified 92.2% eye images and 89.5% non-eye images, yielding an overall accuracy of 89.7%. For the best-effort GLVQ implementation, accuracy may vary due to the non-determism introduced by the enforcement of parallelism for dependent computations. Therefore, we repeated the same experiment 100 times and calculated the average accuracy for best-effort GLVQ. We found that 92.1% eye images and 89.4% non-eye images are correctly classified. The overall accuracy was 89.6% with a very small standard deviation of 0.07%. These results suggest that best-effort computing leads to significant speedup (4.3X over a conventional parallel implementation) with almost no penalty in accuracy.

In summary, the results of applying best-effort computing to the GLVQ training algorithm demonstrate that best-effort strategies can be used to improve parallel scalability by exposing parallelism at a granularity that is much better suited to the underlying platform.

7 Related Work

Concepts closely related to best-effort computing have been explored in various diverse areas of research. In the area of real-time scheduling, best-effort scheduling is often employed as a mechanism for graceful degradation in the presence of overload [15, 2]. A related concept that has been proposed is the imprecise computation model, wherein the an approximate result of acceptable quality is generated whenever the system cannot produce the exact result within the time constraint or deadline [16, 19].

In the area of maintaining cache consistency, sometimes exact synchronization between source data objects and cached copies is not achievable because of bandwidth or other resource constraints. Best-effort synchronization permits the use of stale (out-of-date) copies and it attempts to minimize the overall divergence between source objects and cached copies by selectively refreshing modified objects [20].

The RM applications' inherent error resilient nature can be used as well to design RM specific error resilient system architectures (ERSA) [3]. It combines cheap, unreliable cores together with a small fraction of reliable cores for running system software, controlling application flow, and recovering from errors generated on unreliable cores. The forgiving nature of multi-media applications has also been exploited to achieve improved yield for semiconductor chips that implement these applications [7]. In this context, the underlying chip substrate is viewed as unreliable due to manufacturing process or environmentally induced phenomena, and applications adapt to the unreliable substrate in order to provide an acceptable result or experience to the end user in the presence of errors. This has further led to the concept of error-aware design [14] and manufacturing test [10].

While these efforts are primarily concerned with task scheduling, cache consistency, or error resilient hardware, our focus is on parallel programming models that bridge the computational needs and the capabilities of multi- or many-core platforms. Specifically, we focus on the emerging classes of RM applications that inherently embody the concept of best-effort computing. We study how the parallel programming model can be used to facilitate the separation of an application's computations into guaranteed and best-effort computations. Furthermore, we emphasize the benefits of best-effort computation for parallel execution on multi-core computing platforms.

Various algorithm-specific techniques exist that improve the efficiency of recognition and mining algorithms so as to achieve the same or nearly the same result with less work. For example, efficient algorithms to reduce the amount of computation in K-means clustering have been proposed in [1, 12, 9]. The objective of these algorithms is to reduce the number of iterations required for convergence, or the total number of distance computations performed over the entire execution of the algorithm. In the context of machine learning, on-line learning algorithms have been shown to be very efficient in processing large data sets by reducing the amount of computation required in the training process while achieving comparable accuracy [5]. Our work is different from, and complementary to, these efforts in multiple ways. First, algorithmic techniques typically focus on providing a mathematical guarantee of accuracy or a bound on the approximation error. The techniques employed are often specific to each algorithm considered. We believe that even the best known algorithms can be substantially optimized by employing best-effort computing during their implementation and making empirically driven tradeoffs between performance and quality of the result. We take a broader approach to best-effort computing through the development of parallel programming models and best-effort strategies that are applicable across a wide range of algorithms, without being bound by mathematical analysis or guarantees. Our objective is to provide a programming framework that allows the application designer to easily express the application in terms of guaranteed and best-effort computations, and to evaluate various best-effort strategies without further adding to the complexity of parallel programming.

In summary, while related concepts have been explored in various areas, the specific context in which we explore best-effort computing, namely parallel programming models for recognition and mining applications, distinguishes our work since it requires substantially different concepts and techniques.

8 Conclusions and Future Work

Recognition and Mining applications are emerging as important workloads for future parallel computing platforms. Many RM algorithms exhibit a "forgiving" nature in that they are designed to work with noisy and large-scale input data, their computations are statistical, and some amount of imprecision is acceptable. To exploit this forgiving nature, we have proposed a parallel best-effort programming model that improve performance with acceptable imprecision. We believe that best-effort computing can help close the gap between the explosive computation requirements and the capabilities of the multi- or many-core platforms. We proposed a parallel template for iterative-convergence algorithms that inherently embodies the concept of best-effort computing, and demonstrated its applicability to two representative RM algorithms, namely K-means clustering and GLVQ training. Results show that K-means has a speedup of 3.5X with an accuracy loss of 1%, while GLVQ has more scalable performance with a 4.3X speedup over 8 cores at an accuracy loss of merely 0.1%.

We are extending the library of best-effort strategies to cover more than those discussed in this paper. Our future work will develop tools to help identify converging data structures and bound the accuracy loss. Overall, a framework will be built to test and compare different best-effort computing strategies on a wider set of applications.

9 Acknowledgements

We thank Eric Cosatto who provided us with the histological micrographs that we use to evaluate K-means. We also thank Narayanan Sundaram for helpful discussions.

References

[1] K. Alsabti, S. Ranka, and V. Singh. An efficient k-means clustering algorithm. In *Proceedings of the 1st Workshop on High Performance Data Mining*, Mar. 1998.

[2] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 100–110, 1991.

[3] J. Bau, R. Hankins, Q. Jacobson, S. Mitra, B. Saha, and A. A. Tabatabai. Error resilient system architecture (ersa) for probabilistic applications. In *2007 IEEE Workshop on Silicon Errors in Logic - System Effects*, Apr. 2007.

[4] J. C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Kluwer Academic Publishers, 1981.

[5] L. Bottou and Y. L. Cun. On-line learning for very large data sets: Research articles. *Applied Stochastic Models in Business and Industry*, 21(2):137–151, 2005.

[6] L. Breslau and S. Shenker. Best-effort versus reservations: a simple comparative analysis. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 3–16, 1998.

[7] M. A. Breuer. Multi-media applications and imprecise computation. In *Proceedings of the 8th Euromicro Conference on Digital System Design*, pages 2–7, 2005.

[8] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[9] A. Gursoy. Data decomposition for parallel k-means clustering. In *International Conference on Parallel Processing and Applied Mathematics*, Apr. 2004.

[10] T.-Y. Hsieh, K.-J. Lee, and M. A. Breuer. An error-oriented test methodology to improve yield with error-tolerance. In *Proceedings of the 24th IEEE VLSI Test Symposium*, pages 130–135, 2006.

[11] Intel Corporation. Intel Threading Building Blocks. `http://www.threadingbuildingblocks.org`.

[12] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[13] D. Kriegman and P. Belhumeur. The Yale face database.

[14] F. Kurdahi, A. Eltawil, A. K. Djahromi, M. Makhzan, and S. Cheng. Error-aware design. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 8–15, 2007.

[15] P. Li and B. Ravindran. Fast, best-effort real-time scheduling algorithms. *IEEE Transactions on Computers*, 53(9):1159–1175, 2004.

[16] J. W. Liu, K.-J. Lin, W.-K. Shih, A. C. shi Yu, J.-Y. Chung, , and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[17] J. B. MacQueen. Some methods for classification and analysis of multivariate observation. In *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[18] A. M. Mood, F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. McGraw Hill, 1974.

[19] S. Natarajan. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[20] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 73–84, 2002.

[21] N. R. Pal, J. C. Bezdek, and E. C.-K. Tsao. Generalized clustering networks and kohonen's self-organizing scheme. *Neural Networks, IEEE Transactions on*, 4(4):549–557, Jul 1993.

[22] K. Pearson. On lines and planes of closest fit to systems of point in space. *Philosophical Magazine, Sixth Series*, 2:559–572, 1901.

[23] A. Sato, H. Imaoka, T. Suzuki, and T. Hosoi. Advances in face detection and recognition technologies. 2005.

[24] K. Takahashi and D. Nishiwaki. A class-modular glvq ensemble with outlier learning for handwritten digit recognition. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition*, page 268, 2003.