Parallelizing the Training of the Kinect Body Parts Labeling Algorithm

Mihai Budiu
Microsoft Research
Mihai.Budiu@microsoft.com

Jamie Shotton
Microsoft Research
Jamie.Shotton@microsoft.com

Derek G. Murray
Microsoft Research
derekmur@microsoft.com

Mark Finocchio
Microsoft
markfi@microsoft.com

Abstract

We present the parallelized implementation of decision forest training as used in Kinect to train the body parts classification system. We describe the practical details of dealing with large training sets and deep trees, and describe how to parallelize over multiple dimensions of the problem.

1 Introduction

Human-computer interaction research has long sought a *Natural User Interface* (NUI), which would make computers effectively invisible for their users. Kinect [5] successfully implements a rich NUI in a consumer device that is designed to complement the Xbox 360 games console, by enabling users to control the console using only body gestures and voice control.

The Kinect body tracking software API provides the real-time position of the body joints of each user [7]. In this paper we focus on one stage from the Kinect body tracking vision pipeline: the stage that recognizes and labels body parts in a scene. This stage uses a *decision forest* classifier, which has been described previously [11]; we summarize the essential aspects in Appendix A.

To handle the huge variability of body shapes, clothing, illumination and furnishings in a realistic setting, the classifier must be trained on a very large data set. Since the input data for the classifier comes from essentially a continuous space of 3D objects, building the classifier also entails discovering the most useful features to use for classification. Finally, to achieve acceptable precision the classifier needs to build large decision trees, whose worst-case size is exponential in the tree depth. Consequently, training an instance of the classifier requires a very large number of calculations. In order to accelerate the training process, we have developed a parallel training algorithm, which runs on a cluster of commodity machines.

We have used the DryadLINQ [12] cluster computing platform to implement the training algorithm. DryadLINQ is a programming environment that transforms declarative queries written in LINQ into parallel programs that execute on a computer cluster. LINQ (Language-Integrated Query) is a set of .NET language extensions that can express data-parallel query operators on abstract collections of strongly-typed objects. LINQ generalizes both the MapReduce [2] programming model (offering a richer set of computational primitives) and the database relational algebra (offering a richer data model). DryadLINQ extends LINQ to support large datasets that are distributed across a cluster. The DryadLINQ compiler generates distributed dataflow graphs which are executed by the Dryad [3] distributed execution engine. An example DryadLINQ implementation of decision trees can be found in [4]; however, the implementation discussed in this paper has been tuned specifically for our problem. A commercial version of the DryadLINQ software has been released in beta form as part of HPC Pack 2008 R2 [6].

2 Learning from a massive set of images

We started from a complex but well-tuned C# single-machine implementation, which took approximately one day to train one tree to depth 18 from 10,000 images, using an 8-core machine. A realistic training scenario is several orders of magnitude larger, so we were forced to build a distributed solution.

Our initial attempt at parallelization used MPI. This code proved brittle and more complex compared to the final DryadLINQ version. Because we were not using a highly reliable supercomputer and interconnect, transient failures were common: even a minor incident, such as temporary network congestion, could bring an MPI computation to its knees, wasting hours or days of compute time. This problem only worsened as we attempted to increase the cluster size.

DryadLINQ provided several advantages compared with MPI: (1) a high-level parallel programming language for building distributed computations instead of a simple low-level messaging API, (2) automatic and efficient serialization of complex data types for transport on the network, (3) resource virtualization, which allowed the manipulation of datasets much larger than the collective memory of the available machines, (4) built-in fault-tolerance and checkpointing, and (5) strong integration with C#, which enabled us to reuse most data structures from the sequential implementation and to orchestrate the many parallel and sequential phases of the training in a single program.

Many of the techniques we present are not tied to DryadLINQ as an underlying platform; they could conceivably be implemented on other MapReduce-based runtimes, such as FlumeJava [1] or Pig [9]. The main advantage of DryadLINQ is the tight native integration with .NET, which makes it very easy to implement in a single .NET application complicated workflows composed of many sequential and distributed parts.

2.1 Data representation

The classifier is trained using a large corpus of images that are synthetically generated using a human avatar model, which is in turn driven by data from a motion capture device. For each synthesized scene, the input contains a pair of PNG images: the first encodes the depth (distance from the camera) at each point in the scene, and the second uses pixel colors to encode the ground truth labeling of body parts (Figure 3). These image-pairs are batched into directories, distributed across the cluster, and replicated multiple times to tolerate disk failures. The training pipeline starts with a distributed preprocessing step that combines each image-pair into a single .NET object. During preprocessing, a set of random transformations are applied to the images: flipping on the vertical axis, combining multiple images of single-players to obtain a multi-player image, adding clutter to the background (couches, walls, etc.), and adding noise to emulate realistic depth camera images. The result of this pre-processing is materialized as a large collection of *data units*, distributed evenly on all machines in the cluster.

The training algorithm implementation is generic: it is not specific to our concrete datatypes and problem, being written using an abstraction for an input item, the data unit. A data unit may represent any sensible unit of data required to compute the desired features, (e.g. an image, a volume, or an arbitrary group of data points). Each data point within a data unit is termed an *example* (in our case a pixel). During training, each data unit will be processed independently of all other data units, possibly on different machines in a cluster. A data unit must fit comfortably in memory. While training classifies individual examples independently, computing the features for an example requires access to other examples within the same data unit (Figure 4). The structure of the data and the vast search space of features means that feature responses must be computed on the fly.

The decision tree is represented as a collection[1] containing a single monolithic C# object. The most important part of the tree is the current frontier, which is represented as a sparse set of node indices. The tree is broadcast to all tasks that process data units during a computation round. Although in

[1] All DryadLINQ computations must operate on collections.

theory decision trees can be sparse, in practice our trees are dense enough to imply a complexity exponential in the tree depth.

Note that, although each example is associated with a unique node in the tree frontier, our implementation does not materialize this association at any time. Instead each example is re-associated with the corresponding frontier node by traversing the partial tree from the root for each new frontier computation. Although this work is redundant, it accounts for less than 1% of the running time, and obviates the need to write this information to disk between rounds.

2.2 Parallel training algorithm

Our training algorithm generates a decision tree breadth first, by iteratively splitting nodes on the tree frontier according to the best feature for each node. There are several orthogonal ways to parallelize our training algorithm, by decomposing the problem along different dimensions [10]: (1) partition the data space—i.e. individual data units, (2) partition the search space—i.e. the nodes of the decision tree, and (3) partition the feature space.

Our implementation parallelizes on all three of these dimensions. The computation proceeds in rounds, where round i processes all nodes at depth i in the decision tree and generates a new frontier at depth $i + 1$. The features and data units are read-only. Each round has four phases, which comprise a large number of parallel tasks. In phase 1 the data is scanned to compute normalization constants. In phase 2 each task operates on one data unit, one partition of the feature space and the entire tree (from the previous round). This phase processes each example, builds a four-dimensional histogram, indexed by $(\text{Node}, \text{Feature}, \text{Class}, \{\text{left}, \text{right}\})$. In phase 3 each task operates on one $(\text{Node}, \text{Feature})$ partition of the histogram, and computes the best feature in that partition for splitting each node in that partition (each task is responsible for a subset of the nodes and features). Finally, phase 4 aggregates the information produced by phase two and produces a new tree frontier by selecting the best overall feature for splitting each node, and generating its left and right children.

Figure 1 shows a slightly simplified Dryad execution graph for building one new tree frontier. In this example the features are divided in 2 parts (one for each large rounded rectangle), the data units into 3 groups, and the tree nodes into 4 groups[2]. The feature pruning eliminates all except the best threshold value for each offset pair θ. The core of the execution graph contains two parallel and independent MapReduce jobs (P2-P3) —one for each partition of the feature space—on the same input data set, followed by a merge. In a single MapReduce job, the “barrier” between the map and reduce stages can lead to inefficient cluster utilization. By parallelizing simultaneously on two dimensions (image data and features), the tasks from the concurrent MapReduce computations can be interleaved, which leads to almost 100% cluster utilization.

2.3 Problem scale

A set of realistic parameters for building a useful decision forest are: 3 decision trees, 2000 feature offset values θ and 50 feature thresholds for f_θ (which amounts to trying 100,000 different features), a depth 20 tree (which implies $N < 2^{20}$ tree nodes), 31 body parts and 300,000 images/tree with 300,000 pixels each (sampled down to 2,000 training examples).

Running our algorithm on a 235-machine cluster has an effective parallelism of 140 (defined as the total computation time in all tasks divided by the wall-clock time for training). Training a single tree requires more than 100 effective CPU-days, executing more than 100,000 processes, and moving almost 30 terabytes of data. The training time depends on the amount of memory and the number of cores available on each machine. Figure 2 shows how the efficiency of our implementation improves as the number of training images is increased.

[2] The degrees of partitioning are parameters that are changed for each tree depth; typical values are 4 parts for features and several hundred parts for data units and nodes. Because the cluster resources are virtualized by Dryad, their size does not have to be tied to the size of the computer cluster.

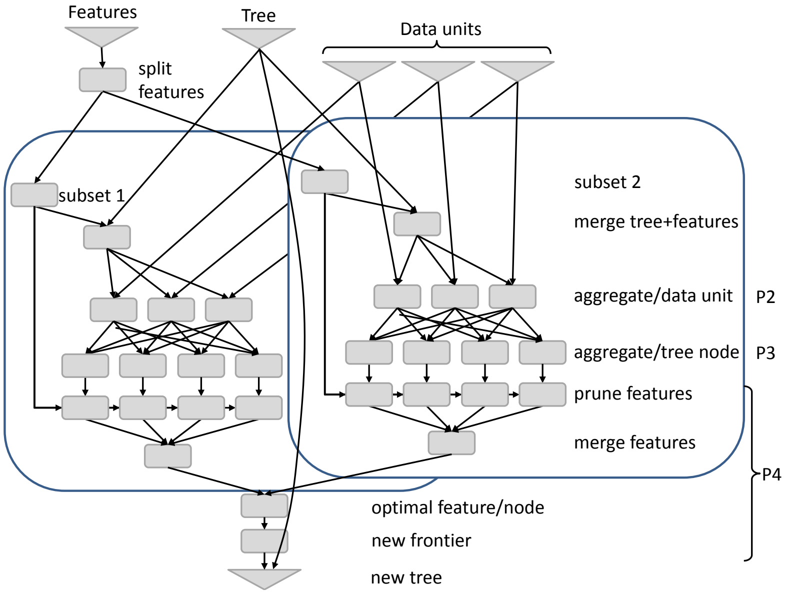

Figure 1: Fragment of a simplified distributed query plan for training one level of the decision tree. The plan has two identical sub-plans, one for each feature subset. P2-P4 correspond to phases 2-4 from the text description.

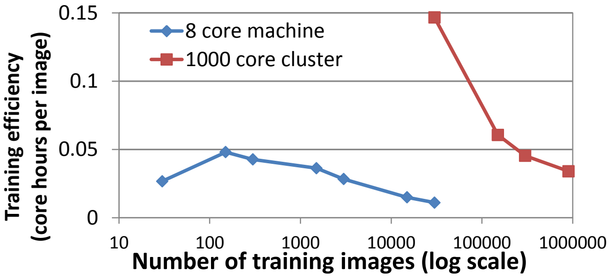

Figure 2: Training Efficiency.

3 Lessons Learned

Our implementation of the training algorithm met or exceeded all expectations in terms of performance and reliability. The training algorithm is not especially I/O intensive, but it is CPU-bound. The multidimensional histograms are sparse, and lazily allocated; a significant amount of time is spent in memory allocation and garbage-collection. Towards the bottom of the tree the partial histogram objects reach sizes on the order of gigabytes each (due to the $O(2^{\text{depth}})$ complexity). At this point some difficult issues surface:

- The I/O layer of DryadLINQ was originally designed to provide high throughput for large collections of small objects. Therefore, it buffers objects and writes them asynchronously to disk. In late tree rounds the buffers fill faster than they can be drained, which leads to thrashing. We modified DryadLINQ to use an adaptive algorithm for buffering, which is sensitive to object size, and falls back to synchronous writes for large objects. Subsequently DryadLINQ has been enhanced with an adaptive buffering algorithm, which manages buffer space dynamically as a function of object sizes, generalizing our approach.

- DryadLINQ attempts to achieve multi-core parallelism by transparently partitioning work between threads in a thread pool. Processing several large partial histogram objects on each thread can artificially make the computation memory-bound. Instead of using the the DryadLINQ-provided multi-core implementation, we parallelized our application across cores at the user-level; our implementation allocates progressively smaller batches as objects grow larger. Another very effective solution is to add more memory to the machines.

- While Dryad provides fault-tolerance for workers, it does not withstand failures in the centralized job manager, which become more likely for long-running jobs. We found it useful to split the training algorithm into multiple Dryad jobs, and to checkpoint the partial tree between jobs. This also allowed us to resume a large job even after discovering and fixing bugs.

We have uncovered some interesting avenues for future research. No distributed filesystem used for manipulating "big data" provides the right APIs to manipulate the kinds of data we used for training: many colocated small images and associated metadata. Our multi-dimensional histogram representations are carefully crafted to balance density and lazy allocation by dynamically changing the representation of some columns from sparse to dense. It would be interesting to generalize this approach to implement a generic distributed data structure, which would manage its own distribution and sparsity dynamically, and adapt to the nature of the stored data. While our adaptive algorithms for buffering and memory allocation are effective at the level of multiple cores on a machine, our tools are currently much more primitive at the cluster level. Finally, the original LINQ code for training decision trees was only on the order of 20 lines of declarative code, but our optimized code is approximately 5× more verbose. We have already used some of the lessons learned to automate some optimizations that we had performed manually [8]. An interesting approach for training would combine both GPU and multi-machine implementations.

Acknowledgments

We would like to thank the following individuals for contributions to this project: Duncan Robertson, Richard Moore, Alex Kipman, Sam Mann, Oliver Williams.

References

[1] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[4] F. McSherry, Y. Yu, M. Budiu, M. Isard, and D. Fetterly. *Scaling Up Machine Learning*, chapter Large-Scale Machine Learning using DryadLINQ. Cambridge University Press, 2011.

[5] Microsoft Corporation. Kinect for Xbox 360. http://www.xbox.com/en-US/kinect, November 2010.

[6] Microsoft Corporation. Announcing the Windows Azure HPC scheduler and HPC Pack 2008 R2 Service Pack 3 releases! http://blogs.technet.com/b/windowshpc/archive/2011/11/11/hpc-pack-2008-r2-sp3-and-windows-azure-hpc-scheduler-released.aspx, November 2011.

[7] Microsoft Research. *Kinect for Windows SDK beta*, July 2011. `http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/docs/ProgrammingGuide_KinectSDK.pdf`.

[8] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, 2011.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data (Industrial Track) (SIGMOD)*, Vancouver, Canada, June 2008.

[10] F. J. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 3:131–169, 1999.

[11] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from a single depth image. In *Computer Vision and Pattern Recognition (CVPR)*, Colorado Springs, June 21-23 2011.

[12] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)*, page 14, 2008.

A Body Parts Classifier

The Kinect body parts classifier is a randomized decision forest, built using fully-supervised training [11]. The classifier is presented with an image in depth space (in which each pixel indicates distance from the camera), and produces—for each pixel—a distribution over 31 body parts (Figure 3). The classifier uses only one type of feature, described below. Building the classifier entails searching a large randomly-generated space of possible features. To alleviate overfitting, the classifier uses a set of three decision trees, each trained on a random subset of pixels and input images. The decision forest is produced by training three independent decision trees in sequence; this paper only discusses the training of a single tree.

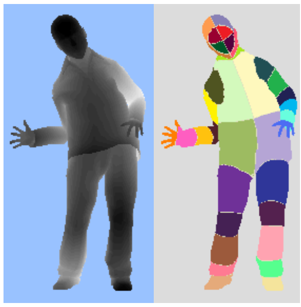

Figure 3: Training data: a pair of depth map and labeled body parts.

The features used are simple depth comparisons. At a given pixel $\mathbf{x}$, the features compute

$$f_\theta(I;\mathbf{x}) = d_I\left(\mathbf{x} + \frac{\mathbf{u}}{d_I(\mathbf{x})}\right) - d_I\left(\mathbf{x} + \frac{\mathbf{v}}{d_I(\mathbf{x})}\right) \tag{1}$$

where $d_I(\mathbf{x})$ is the depth at pixel $\mathbf{x}$ in image I, and parameters $\theta = (\mathbf{u};\mathbf{v})$ describe offsets $\mathbf{u}$ and $\mathbf{v}$. The normalization of the offsets by $\frac{1}{d_I(\mathbf{x})}$ ensures the features are depth invariant: at a given point on the body, a fixed world space offset will result whether the pixel is close or far from the camera. The features are thus 3D translation invariant (modulo perspective effects). Figure 4 illustrates two features at different pixel locations $\mathbf{x}$. Feature f_{θ_1} looks upwards, and the formula will give a large positive response for pixels $\mathbf{x}$ near the top of the body, but a value close to zero for pixels $\mathbf{x}$ lower down the body. Feature f_{θ_2} may help find thin vertical structures such as the arm.

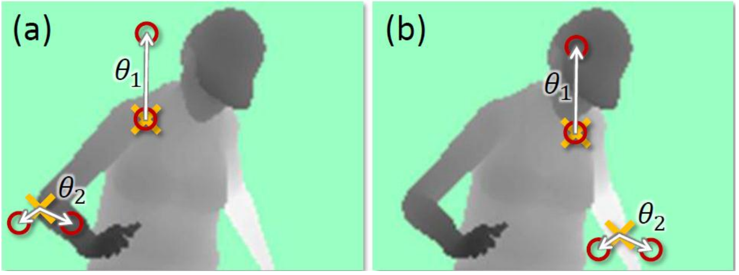

Figure 4: Depth image features. The yellow crosses indicates the pixel $\mathbf{x}$ being classified. The red circles indicate the offset pixels. In (a), the two example features give a large depth difference response. In (b), the same two features at new image locations give a much smaller response.