Automatic Parallelization of Canonical Loops

Leonardo L. P. da Mata, Fernando Magno Quintão Pereira, Renato Ferreira

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte – MG – Brazil

{barroca,fpereira,renato}@dcc.ufmg.br

Abstract. *This paper presents a compilation technique that performs automatic parallelization of canonical loops. Canonical loops are a pattern observed in many well known algorithms, such as* frequent itemsets, K-means and K nearest neighbors. *Automatic parallelization allows application developers to focus on the algorithmic details of the problem they are solving, leaving for the compiler the task of generating correct and efficient parallel code. Our method splits tasks and data among stream processing elements and uses a novel technique based on labeled-streams to minimize the communication between filters. Experiments performed on a cluster of 36 computers indicate that, for the three algorithms mentioned above, our method produces code that scales linearly on the number of available processors. These experiments also show that the automatically generated code is competitive when compared to hand tuned programs.*

1. Introduction

Multi-core computing is becoming industry's standard choice towards high performance. Examples of new multi-core machines are Sun Ultra Sparc T1 [Kongetira et al. 2005], Sony/IBM/Toshiba Cell [Hofstee 2005] and NVIDIA GeForce 8800 GTX [Ryoo et al. 2008]. Furthermore, multi-core settings are also more and more present in the personal computer market, given that Intel and AMD are now distributing machines with two and four cores as standard selling options. This proliferation of parallel hardware is driving a true revolution in the programming language community. In the words of Hall *et al.* [Hall et al. 2009], "the next fifty years of compiler research will be mostly devoted to the generation and verification of parallel programs". As an illustration, 17 out of 34 papers published on the proceedings of the 2008 ACM's Conference on Programming Languages, Design and Implementation [Gupta and Amarasinghe 2008] deal directly with concurrent programs.

Multi-core computers give the application developer the opportunity to solve problems in parallel; however, designing and coding concurrent programs is still a difficult task. This difficulty stems from the fact that the developer must not only reason about how to split his problem into smaller chunks that can be solved in parallel, but also worry about classic concurrent hazards such as data races and deadlocks. Thus, there exists a large research effort in the programming language community to move this burden from the programmer to the compiler. A compiler that performs automatic parallelization of sequential programs allows the application developer to concentrate on the algorithmic details of the target problem, while still taking benefit from the parallel hardware.

We have developed a compilation technique that automatically parallelizes *canonical loops*. Canonical loops, defined in Section 2.1, are a recurring

pattern in many algorithms that iterate over some amount of changing data. This pattern constitutes the core of algorithms as diverse as frequent itemsets [Agrawal et al. 1993], K-means [Steinhaus 1956, Macqueen 1967], K nearest neighbors [Witten and Frank 2002], Ransac [Fischler and Bolles 1981] and many variations of genetic algorithms [Banzhaf et al. 1998]. We obtain parallelism by translating canonical loops into sequences of filters that run on Anthill [Ferreira et al. 2005]. Anthill is a data-flow based runtime framework that supports the development and testing of high performance parallel applications. We chose this tool because it allows us to integrate into a single computing body a large cluster of heterogeneous commodity hardware.

We have developed a source-to-source compiler that translates sequential C programs to C programs written using the Anthill libraries. We have implemented our technique using the back-end provided by the Cetus compiler [Johnson et al. 2004], and the front-end provided by the ANTLR parser [Parr and Quong 1995]. To validate our method, we have parallelized three different algorithms: K-means, frequent itemsets and K nearest neighbors. Our experiments were performed on a cluster of 36 computers. We have observed that the parallel programs produced using our compiler achieve linear speedups on the number of available filters. Furthermore, a comparison between the generated code and equivalent, previously existing hand-crafted programs reveal that our strategy can deliver performance that matches that obtained by seasoned programmers.

The rest of this paper is organized as follows: Section 2 defines concepts that will be used throughout the paper, such as canonical loops and stream programming. In Section 3 we show how we translate three classic data-mining algorithms written using map-reduce operations to compositions of filters. Section 4 explains our compilation technique with greater details. We show experimental results that validate our approach in Section 5. Finally, Section 6 concludes this paper.

2. Background

In this section we define the main concepts used throughout this paper, namely, canonical loops, map-reduce operations, stream programming and the Anthill framework.

2.1. Canonical Loops

Algorithm 1 shows the general form of a canonical loop. *MR* is a *map-reduce* operation, that is, it is written as a *combination* of two operations - *map* and *reduce* - applied on a polynomial data structure [Morita et al. 2007, Akimasa et al. 2009]. The function *map* applies a function f on every element of an input data structure, normally a list, i.e:

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

The function *reduce* collapses a data structure into a single value via repeated applications of a binary associative operator $\odot$ on its elements:

$$reduce\ \odot\ [x_1, x_2, \ldots, x_n] = x_1 \odot x_2 \odot \ldots \odot x_n$$

Map and reduce, as previously defined, are *parallel skeletons* [Cole 1988]. Because $\odot$ is associative, combinations of map and reduce can be implemented efficiently in parallel. An operation such as $map\ f\ (reduce\ \odot\ L_a)\ L_b$ is $O(|L_a| \times |L_b|)$ in a sequential machine,

Algorithm 1 – Canonical Loop:

1: $L_1 \leftarrow \ldots$
2: **while** ... **do**
3: $\quad L_2 \leftarrow MR_2(L_1)$
4: $\quad L_3 \leftarrow MR_3(L_2)$
5: $\quad \ldots$
6: $\quad L_n \leftarrow MR_n(L_{n-1})$
7: $\quad L_1 \leftarrow L_n$
8: **end while**

whereas $reduce\ \odot\ (map\ f)\ L_a$ is $O(|L_a|)$; however, both are $O(\log |L_a|)$, given a polynomial number of processors. As an illustration, map-reduce operations model Google's Map-Reduce framework for distributed computing [Dean and Ghemawat 2004].

The First Homomorphism Theorem [Gibbons 1996] says that a homomorphism on lists can be written as the composition of reduce and map. The function h is a list homomorphism if there exists an operator $\odot$ such that, given the concatenation operator $+\!\!+$, we have:

$$h(x\ +\!\!+\ y) = h(x) \odot h(y)$$

Morita *et al.* have shown how to derive list homomorphisms from sequential algorithms automatically [Morita et al. 2007]. We are solving a different problem. We already assume that our *MR* operations use associative operators. Our great contribution is how to split the work of *MR* operations among processors, and how to implement the communication between processors efficiently. The parallel SML compiler (PSML) of Scaife *et al* [Scaife et al. 2005] is the closest work to our research. PSML derives parallel OCaml code from map and reduce skeletons present in sequential ML programs. The main difference between Scaife's approach and ours is that we are targeting an execution environment where communication is much more costly. Thus, in addition of partitioning compositions of maps and folds into separate processes — which are stream filters in our case — we are also trying to minimize the communication between different processing elements via labeled-stream channels, a concept that we describe in the next Section.

A short digression for notation's sake In this paper we use a functional notation to describe our target algorithms, in order to make it easier to see map-reduce patterns in the sequential applications that we compile. However, we are compiling C code, and thus, our compiler recognizes map-reduce patterns implemented as imperative loops. For instance, given the pattern in Equation 1, we search for imperative constructions such as the C program in Figure 1.

$$L_x \leftarrow \text{map}\ (\lambda b\ .\ \text{reduce}\ (f_r\ b)\ L_a)\ L_b \tag{1}$$

We point out that the compilation techniques described in this section are valid independent on the notation used to represent the sequential programs.

2.2. Stream Programming

Quoting Spring *et al.*, "The stream programming paradigm aims to expose coarse-grained parallelism in applications that must process continuous sequences of

```
int Lx[NB]; int Lb[NB] = ...; int La[NA] = ...;
for (ib = 0; ib < NB; ib++) {
  Lx[ib] = InitialValue(Lb[ib]);
  for (ia = 0; ia < NA; ia++)
    Lx[ib] = fr(Lx[ib], La[ia]);
}
```

Figure 1. Map-reduce operation written in C, representing Equation 1.

events" [Spring et al. 2007]. In this paradigm, programs are modeled as compositions of filters, which communicate via uni-directional channels. Each filter is a processing unit, which may or may not keep an internal state, and that reads data from input channels and writes data on output channels. The output of a filter is uniquely determined by the contents of its input channels, and by its internal state, when it exists. Stream programming facilitates the development of parallel applications because filters, being independent on each other, may be scheduled to execute in parallel, without the developer having to worry about problems such as data races.

Anthill [Ferreira et al. 2005] is a runtime framework for programming data-flow applications. In this model, applications are decomposed into a set of filters connected via streams. Streams are communication abstractions that provide unidirectional pipes to carry data from one filter to another. Anthill is composed of two parts: (i) a middleware that coordinates the execution of stream programs distributed across a cluster of heterogeneous commodity machines; and, (ii) a library, implemented in C, that provides the developer of parallel applications with functionalities to write filters and streams. At runtime, filters can be instantiated multiple times, thus achieving data parallelism as well as task parallelism during execution. Streams, therefore, are capable of transferring messages from a set of filter instances to another set of filter instances. Anthill defines three communication patterns that a stream can use: *broadcast*, *round-robin* and *labeled-streams*. These communication models are explained below. In every case, we assume that a filter f_o is connected through n communication channels to n filters $f_{di}, 1 \leq i \leq n$.

- **Broadcast:** each message produced by f_o is replicated n times, and sent through each of the n channels. The flow of messages never changes during the execution of the application.
- **Round-robin:** each message produced by f_o is sent through only one of the n possible channels. A different channel is chosen for each message until all of them are used. This pattern is then repeated, in a round-robin fashion.
- **Labeled-stream:** each message produced by f_o is associated to a label, and sent through some of the n possible channels, which is chosen by a hash function that maps labels to channels. The hash function is created statically, during the compilation of the Anthill application, and it never changes at runtime.

3. Examples

In this section we show how three different algorithms — frequent itemsets, K-means and K nearest neighbors — are mapped to Anthill filters. Due to space constraints, we will be using very simplified descriptions of each algorithm. Full blown implementations in SML, plus equivalent C codes, are available at `http://www.dcc.ufmg.br/~barroca/parallel`.

3.1. K-means

K-means is a clustering algorithm used to separate a set of input objects into K clusters [Macqueen 1967, Steinhaus 1956]. In this example, we assume that these objects are points in the Euclidean space. The input of the algorithm is a list L_p of points that must be grouped into clusters, plus an initial list L_c of K *centroids*. The centroids are also points in Euclidean space. A point $p \in L_p$, closest to $c \in L_c$, is said to be part of the cluster c. We assume that the initial position of the centroids is chosen randomly. During the execution of the algorithm, each centroid will converge to a position that tends to maximize the number of points that are part of its cluster. The algorithm terminates when either the centroids stop changing position, or a maximum number of iterations is reached. A simplified version of K-means is described in Algorithm 2. The algorithm contains two map-reduce operations. The first one, in line 6, maps each point $p \in L_p$ to a tuple (p, c), where c is p's nearest centroid. The second operation traverses the list of centroids L_x, and for each centroid c, the reduction traverses the list of tuples L_t. Given an element $(p, c') \in L_t$, whenever $c = c'$, the value of p is used to updated the position of c.

Algorithm 2 – K-means:

```
1: L_p ← list of points
2: L_c ← initial centroids
3: L_x ← ∅
4: while L_x ≠ L_c do
5:     L_x ← L_c
6:     L_t ← map(λp . reduce (minDistanceFromPointP p) L_x) L_p
7:     L_c ← map(λc . reduce (updateCentroidIfEqualToC c) L_t) L_x
8: end while
9: return L_c
```

Figure 2 shows the filters that our compiler produces for Algorithm 2. In this, and the next examples, we will be using the simplified notation of Figure 2 (b) to describe stream applications running on Anthill. We have generated two kinds of filters, that we call F_{sep} and F_{upd}. F_{sep} separates points according to the nearest centroid. Thus, given a set of N points $\{p_1, \ldots, p_N\}$, plus a set of K centroids $\{c_1, \ldots, c_K\}$, F_{sep} produces a set of N tuples $\{(p_1, c_i), \ldots, (p_N, c_j)\}$, where each point is bound to its closest centroid. Filter F_{upd} updates the positions of the centroids given the positions of the points that compose its cluster. When implementing the first map-reduce operation, in line 6, the nature of the K-means algorithm allows us to split the points equally among the instances of the filters F_{sep}, but these filters must still read the full set of centroids. Thus, points are sent to those filters via round-robin, whereas centroids are sent via broadcast. Similarly, we can split the centroids equally among filters F_{upd}, but the filter that is in charge of updating centroid c must have access to all the points bound to c. Thus, the way data is sent to filters F_{upd} depends on information produced at runtime. That is, the centroid assigned to a point determines the filter F_{upd} where that point will be sent. This is a case of labeled stream communication, where the message is a point, and its label is the centroid assigned to that point. There is only one filter f_{upd}, instance of F_{upd}, in charge of updating a given centroid c. If a point p is clustered around c, i.e, there is a tuple $(p, c) \in L_t$, then p will be sent only to f_{upd}.

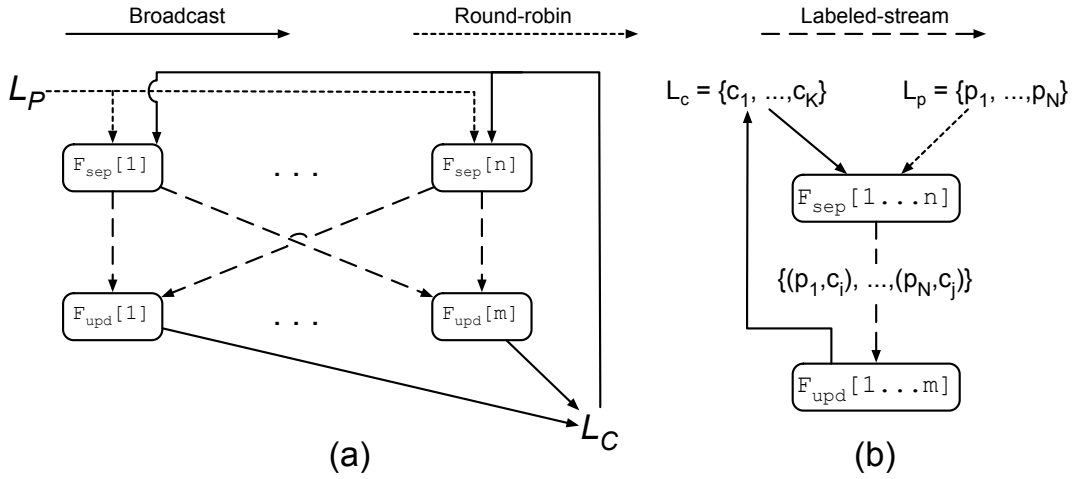


Figure 2. (a) K-means represented as a collection of filters. (b) Simplified view.

3.2. Frequent Itemsets

The frequent itemsets algorithm is used to discover association rules in large datasets [Agrawal et al. 1993]. In this paper, we used the version of the algorithm known as *apriori*. The input of the algorithm is an integer number σ, called the support value, plus a set of transactions $L_t = \{t_1, t_2, \ldots t_n\}$, where each transaction is itself a set of attributes, that is, $t_i = \{a_1, a_2, \ldots, a_m\}$. Transactions may have different cardinalities. The objective of the algorithm is to discover which subsets of attributes occur in more than σ transactions. A description of this technique is given in Algorithm 3. The canonical loop is composed of three sequences of reductions. The first, given in line 10, binds candidates (c) to the transactions (t) where they occur, producing a list L_n of tuples (c, t) [1]. The second, showed in line 11, counts how many times each candidate occurs, i.e, it goes through the list of candidates, and for each candidate c, it traverses L_n, and sums the tuples that match (c, t), for some $t \in L_t$. It also separates those candidates that are frequent, that is, that appeared more than σ times. The third reduction produces a new list of candidates to be verified. New candidates of cardinality N are produced by pairing old candidates of cardinality $N - 1$ with frequent attributes. For the sake of simplicity, our specification omits obvious optimizations.

The left side of Figure 3 outlines the filters that our compilation technique produces for Algorithm 3. In the first phase of the algorithm, each transaction can be processed independently on the others, but the processing of one transaction requires all the candidates. Thus, candidates are fed to instances of filters F_{cnt} by broadcast, whereas transactions are passed via round-robin. For each of the incoming transactions, a filter produces a pair (c, t), where c is a candidate that is frequent in transaction t. This data is then sent, via labeled streams, to filters F_{sup}, that sum all the occurrences of a given candidate, passing that candidate forward if, and only if, that number is greater than the support. The label used is the candidate, as the summation filter must be able to see all the occurrences of a given candidate.

[1] In fact, this operation produces a list of lists of tuples. We omit the operation used to concatenate these lists into a single data structure

Algorithm 3 – Frequent Itemsets:

1: $\sigma \leftarrow$ support value
2: $L_t \leftarrow$ list of transactions
3: $L_a \leftarrow$ attributes that occur more than σ times among transactions
4: $L'_c \leftarrow L_a$
5: $L_{res} \leftarrow \emptyset$
6: $L_c \leftarrow \emptyset$
7: **while** $L_c \neq L'_c$ **do**
8: $\quad L_{res} \leftarrow L_{res} +\!\!+ L'_c$
9: $\quad L_c \leftarrow L'_c$
10: $\quad L_n \leftarrow$ map $(\lambda t$. reduce (getCandidatesThatAreInT $t)$ $L_c)$ L_t
11: $\quad L_f \leftarrow$ removeUndefs (map $(\lambda c$. if (reduce (countNumberOfC $c)$ $L_n) > \sigma$ then c else $\perp)$ $L_c)$
12: $\quad L'_c \leftarrow$ map $(\lambda c$. reduce $(\lambda a$. $a \cup c)$ $L_a))$ L_f
13: **end while**
14: **return** L_{res}

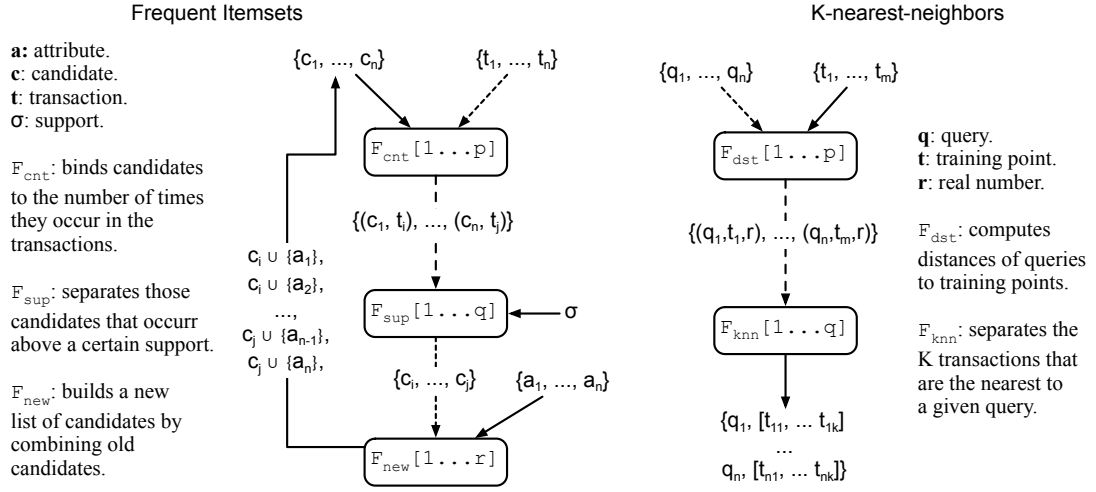


Figure 3. (Left) Frequent itemsets as stream program. (Right) K nearest neighbors as stream program.

3.3. The K Nearest Neighbors

The algorithm called K nearest neighbors is a technique [Witten and Frank 2002] that classifies elements from a set L_q of *queries* based on the K nearest elements from a set L_t of training data. This method is described in Algorithm 4. Algorithm 4 is a *degenerate canonical loop*, that is, it contains two map-reduce operations, but those will be executed only once. The first operation, in line 3, contains only mapping calls; for each query q it produces a list of $|L_t|$ tuples $(q, t, r), t \in L_t$, where r is the distance from q to t. The second operation, given in line 4, finds the K training points closest to q.

The filters that we produce for this application are shown on the left side of Figure 3. Our compiler generates two kinds of filters, F_{dst}, which computes the distance from each query to each training point, and F_{knn}, which finds the K nearest training points to

Algorithm 4 – K Nearest Neighbors:

1: $L_q \leftarrow$ List of queries
2: $L_t \leftarrow$ Training set
3: $L_d \leftarrow$ map $(\lambda q$. map $(\lambda t$. $(q, t,$ distance(q, t))) $L_t)$ L_q
4: $L_k \leftarrow$ map $(\lambda q$. reduce (findTheKClosestTraningPointsToQ $q)$ $L_t)$ L_q
5: **return** L_k

each query, given the distances computed before. The communication between F_{dst} and F_{knn} happens via labeled streams, as we can find the training points that are the closest to a given query independent on the other queries. Each message (q, t, r) is labeled by the query q, and it contains a training point t, plus the distance r between t and q.

4. Automatic generation of parallel code

We have implemented a source-to-source compiler that maps sequential programs, written in C, to C programs that use the filter-stream libraries provided by Anthill. Our compiler has two main tasks: (i) split the target application into filters, and (ii) decide how these filters communicate. The compilation process can be entirely automatic, but can also be directed by the developer. Our compiler is able to recognize some map-reduce patterns in C code, and generate filters for them; however, our compiler does not guarantee that every map-reduce operation is found. For these cases, we provide developers with annotations, implemented as C macros, to explicitly point where the reductions are. The Cetus compiler contains over 74,000 lines of commented Java code, and our modifications on its back-end account for about 6,400 extra commented lines, divided as follows: conversion to SSA-form, 1,700 lines; static analyses to discover map-reduce patterns, 3,400 lines; printing of filter programs, 1,300 lines.

On the number and distribution of filters Our compiler creates one type of filter for each map-reduce operation present on the canonical loop. That is, given a program such as Algorithm 1, with n map-reduce operations, we generate n different types of filters. During the execution of the parallel application produced, each type of filter may have many instances running concurrently. The number of instances of each filter is defined by the application developer, before the execution of the parallel application. For details on how to implement synchronization and communication between different processes representing *the same* map-reduce operation, see Scaife *et al* [Scaife et al. 2005].

The Anthill framework takes care of the deployment of filters across the cluster of available machines, but the assignment of filters to machines is performed by our compiler. Although there are strategies for optimizing the distribution of filters throughout the cluster [Góes et al. 2005, do Nascimento et al. 2005], we opted to only allow two possible placement strategies: *implicit* and *explicit*. In the first case, our compiler randomly chooses a machine to run each filter. In the second, the user defines the placement.

Defining the communication between filters representing different map reduce operations We use a standard reaching definition analysis [Aho et al. 2006] to determine which data must be communicated between filters. For instance, considering Algorithm 2, our reaching-definition analysis reveals that the data structures L_x and L_p are necessary to the filters created for the map-reduce operation in line 6.

Once the reaching-definition analysis has found which data must be passed across filters, the next step of compilation is to determine *how* this data must be communicated. As we have explained in Section 2.2, there are only three possible kinds of communication channels between filters: broadcast, round-robin and labeled-stream. The first two methods are the standard choice of our compiler, whereas labeled-stream is used as an optimization, whenever possible.

Given a pattern like that in Equation 1, Section 2.1, a filter derived from this map-reduce operation will receive the data-structure L_a via broadcast, and data from L_b via round-robin. The code that we produce for this example is outlined in Algorithm 5. The functions RCVBC and RCVRR implement a broadcast communication channel, and a round-robin communication channel, respectively. The unit values ι are used to initialize the results of the map-reduction.

Algorithm 5 – Default parallel code automatically produced:

```
1: L_x ← [ι, ..., ι]
2: L_a ← RcvBc()
3: while b ← RcvRr() do
4:     L_x[b] ← reduce f_r L_a
5: end while
```

The default strategy allows us to distribute the work performed on L_b equally among the available instances of the filter. However, we have an even more aggressive parallelization technique, that is based on the fact that many of our reduction functions follow the pattern given below, where $\oplus$ is any associative operator:

$$f_r = \lambda a\ .\lambda(b, a_{acc})\ .\ \text{if } a \subseteq b \text{ then } (b \setminus a) \oplus a_{acc} \text{ else } a_{acc} \tag{2}$$

For instance, function updateCentroidIfEqualToC in Algorithm 2 is defined as follows:

$$\lambda c\ .\ \lambda((c', p), (c_{acc}, n))\ .\ \text{if } c = c' \text{ then } ((n \times c_{acc} + p)/(n+1), (n+1)) \text{ else } (c_{acc}, n)$$

The function above is initialized with $(c_{acc}, n) = (c_\iota, 0)$, where c_ι is the null centroid with all coordinates equal to zero. This function keeps a counter n of centroids already processed, to weight the new average centroid, every time a new value must be computed. Independent on the way centroids are updated, the essential point here is that *only the points that are clustered to a given centroid have influence on its updating*. That is, to update the coordinates of a centroid c, we do not need to look at every input point, only those that are closer to c. Thus, the filter responsible for the updating of a given centroid c must receive only those points that have been clustered around c.

In general, we do not have to send the whole L_a data structure seen in equation 1 via broadcast to every instance of the filter responsible for that map-reduction. Instead, we split the data from L_a among the filters, using a hash function, and implement communication channels based on labeled streams. The filter produced to receive labeled data contains only the computation in the "then" branch of the reduction function, that is: $\lambda a\ .\ \lambda(b, a_{acc})\ .\ (b \setminus a) \oplus a_{acc}$. The general form of a filter that uses labeled stream is given in Algorithm 6, where RCVLS is a labeled-stream channel. For each tuple (b, a), where

Algorithm 6 – Parallel code for Label-Stream pattern:

1: $L_x \leftarrow [\iota, \ldots, \iota]$
2: **while** $(\texttt{label, value}) \leftarrow$ RCVLS() **do**
3: $\quad L_x[\texttt{label}] \leftarrow L_x[\texttt{label}] \oplus \texttt{value}$
4: **end while**

Figure 4. Comparison between parallel code produce with and without the labeled stream optimization.

a) Original, sequential code

```
int Lb[NB] = ...;
int La[NA] = ...;
int Lx[NB];
for (ib = 0; ib < NB; ib++) {
  Lx[ib] = InitialValue(Lb[ib]);
  for (ia = 0; ia < NA; ia++) {
    if (Subset(La[ia], Lb[ib])) {
      Lx[ib] = fr(Lx[ib], Dif(La[ia], Lb[ib]));
    }
  }
}
```

b) Parallel code without the labeled-stream optimization

```
int *Lx = InitArray();
int *La = RcvBc();
RRData* bdt;
while (bdt = RcvRr()) {
  for (ia = 0; ia < NA; ia++) {
    if (Subset(La[ia], bdt)) {
      Lx[bdt] =
        fr(Lx[bdt], Dif(La[ia], Lb[ib]));
    }
  }
}
```

c) Parallel code otimized with labeled streams

```
LSData* adt;
while (adt = RcvLs()) {
  Lx[adt->label] =
    fr(Lx[adt->label], adt->value);
}
```

a and b are the parameters of the reduction function in Equation 2, we create a message with two fields: a label, formed by b, and a value, formed by the difference $b \setminus a$.

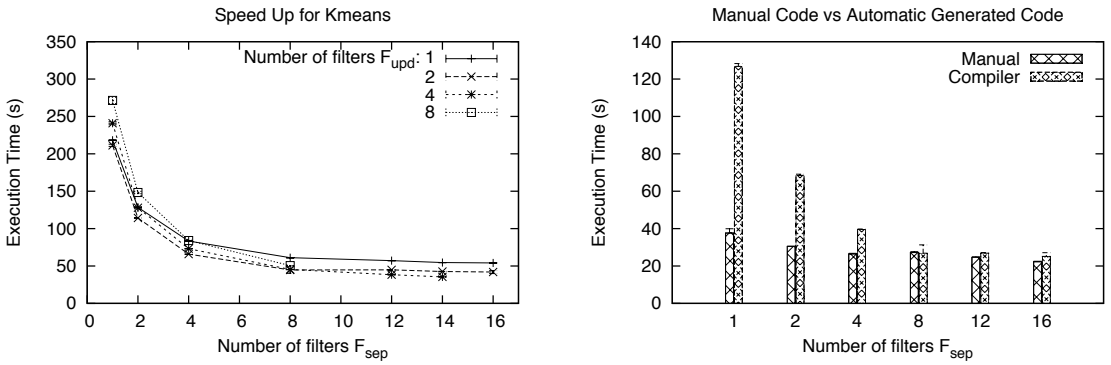
Figure 4 illustrates the labeled-stream optimization. Figure 4 (a), shows a map-reduce loop, written in C, containing the pattern described in Equation 2. Figure 4 (b) shows the equivalent parallel code produced by our compiler when the labeled-stream optimization is disabled. Figure 4 (c) shows the code optimized with labeled-streams.

The next question that must be addressed is how to route labeled data among filters. Routing is performed by a hash function h, created at compilation time, that maps values from the L_a set (Equation 1) to filters. Thus, given a data $a \in L_a$, where $\exists b \subseteq a, b \in L_b$, we have that $h(b) = s$, is the filter where message a must be delivered.

5. Experiments

In order to validate our compiler, we have used it to automatically parallelize the three algorithms described in Section 3. We have performed two types of experiments. The first type shows the amount of speed up that we obtain through the parallelization of the code. The objective of these experiments is to show that our automatically generated code presents good scalability when faced with an increasing number of processors. The second type compares the performance of the code that we produce against similar programs that have been implemented by hand. The objective of this second set of tests is to show that our compiler delivers code that is competitive with the code produced by seasoned programmers. All the hand-crafted programs are available at `http://www.dcc.ufmg.br/~barroca/parallel`. The implementations of k-means and frequent itemsets are the same programs presented in [Ferreira et al. 2005].

Figure 5. K-means. (Left) speed-up due to increase in number of filters. (Right) comparison with program written by hand.



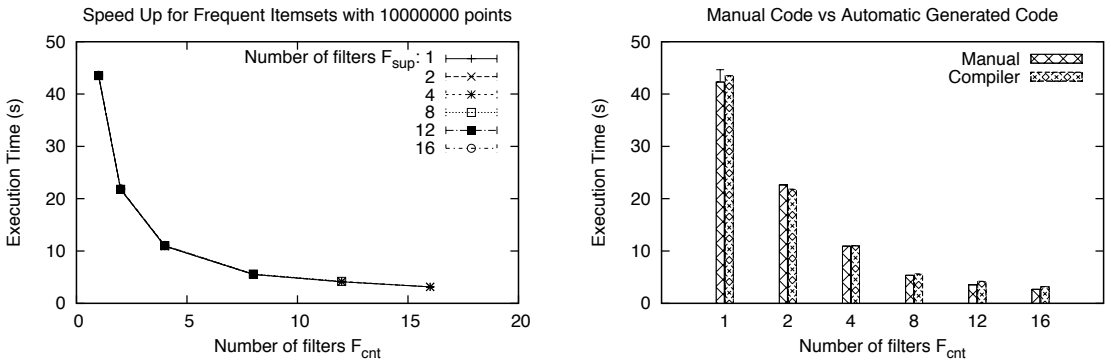
KNN was implemented by one of this paper's authors.

The experiments were performed in a cluster formed by 36 commodity computers. Each of our computers is an AMD Athlon 64 3200+ (2GHz) processor with 512KB of cache, 2GB of RAM and 160GB SATA disks (ATA 150). The computers are connected through a Gigabit Ethernet. The equipment was exclusively dedicated to our experiments, that is, besides our programs, each computer would be running only system processes. The operating system used is Debian GNU / Linux 4.0, kernel version 2.6.18-6. The version of Anthill used was 3.1.

K-means Figure 5 shows the results of our experiments for K-means, the algorithm described in Section 3.1. For this experiments, we are using 200,000 points and 2,000 centroids. The left graph depicts the behavior of the parallel program produced by our compiler in face of a changing number of filters. The chart shows that the performance of this application depends heavily on the number of instances of the filters F_{sep}, which is responsible for matching points to the nearest centroids. For instance, by going from one filter F_{sep} to two, we get a speed-up of 1.69 times. this speed-up factors tend to decrease with the increase on the number of filters, if the size of the input remains constant. Going from 8 to 16 filters gives a speed-up of only 1.11 times. However, we get linear speed-ups if the number of filters increase in proportion to the input size. That is, the overall execution time remains constant when we double both the number of input points and the number of instances of F_{sep}. We notice that the number of instances of filters F_{upd}, which updates the location of each centroid c, given the points clustered around c, has very small impact on the overall application performance. This is due to the fact that points are not clustered uniformly around centroids.

The right side of Figure 5 compares the code produced by our compiler with a manual implementation of K-means that uses the Anthill libraries. We notice that for a small number of filters, the handcrafted program greatly outperforms our parallel code; however, as the number of processors increase, both implementations converge to similar execution times.

Figure 6. Frequent Itemsets. (Left) speed-up due to increase in number of filters. (Right) comparison with program written by hand.



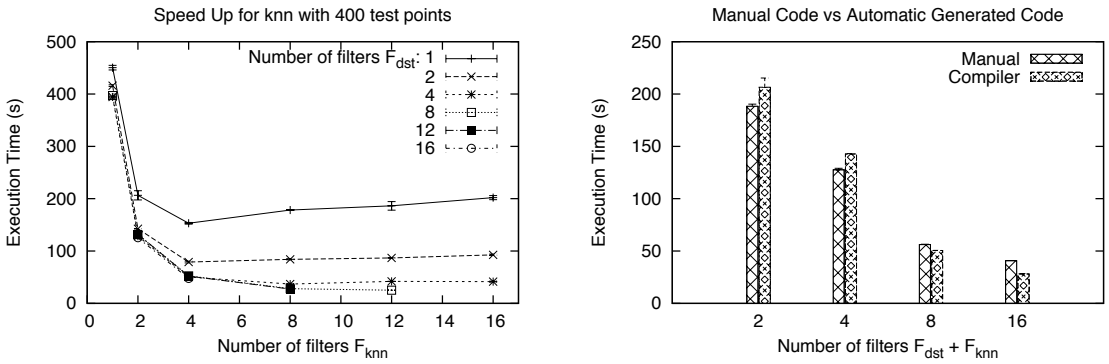
Frequent Itemsets Figure 6 shows the gains that we obtain by parallelizing the frequent itemsets algorithm described in Section 3.2. The experiments were performed with an input of 10^7 random candidate points, each point containing up to 20 dimensions. The left chart shows how the increase in the number of filters affect the execution time of the algorithm. These experiments use different numbers of filters F_{cnt} and F_{sup}, and one single instance of filter F_{new}. We notice that the performance of our parallel application depends only on the number of instances of the filter F_{cnt}, which counts how many times each candidate occurs among the input points. The time complexity of this phase of the algorithm is $O(C \times P)$, where C is the number of candidates, and P is the number of input points. In these experiments the execution time does not depend on the number of instances of the other types of filters, because these filters are responsible for phases of the algorithm that have much lower complexity than the first phase. That is why the six different runs with a varying number of instances of filters F_{sup} fall on the same line.

The chart on the right side of Figure 6 compares the program produced by our compiler with the implementation used in [Ferreira et al. 2005]. We point that, although the hand-crafted code is heavily optimized, our parallel program presents very similar execution times, although it is produced automatically.

K-Nearest-Neighbors Figure 7 outlines the performance behavior of the KNN algorithm described in Section 3.3. In this experiments, our input consists of 10,000 training points, and 400 queries. Contrary to K-means, both types of filters used in the implementation of KNN present good scalability. This behavior is due to two main reasons. First, the semantics of the algorithm allows our compiler to divide work equally among all the types of filters. Second, there are no iterations in this case, and thus, no necessity of synchronizing data between different filter types. As an illustration, given 16 instances of filters F_{knn}, we obtain almost perfect speed-ups - close to $\times 2$, when doubling the number of instances of filter F_{dst}.

The comparison with the program written by hand is very promising. Our automatically produced code is slight slower than the hand-crafted implementation, when one instance of each kind of filter is present. As we double the number of instances of

Figure 7. KNN. (Left) speed-up due to increase in number of filters. (Right) comparison with program written by hand.



each type of filter, we observe that our parallel code is able to outperform the manually produced program when there are eight and 16 instances of each type of filter.

6. Conclusion

In this paper we have presented an automatic parallelization technique that relies on stream filters to obtain performance gains. Our experiments have shown that our method generates efficient programs, that compete even with hand-coded applications. Our compiler finds parallelism by searching for map-reduce patterns in the source code, and recognizing conditions that allow it to wire more efficient communication channels between filters. We believe that our most important contribution is the fact that we allow the application developer to produce high performance parallel code without having to worry about details typical of parallel algorithms, such as race-conditions and thread synchronization.

References

Agrawal, R., Imielinski, T., and Swami, A. N. (1993). Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

Akimasa, Matsuzaki, K., Hu, Z., and Takeichi, M. (2009). The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In *POPL*, pages 177–185. ACM.

Banzhaf, W., Francone, F. D., Keller, R. E., and Nordin, P. (1998). *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann.

Cole, M. I. (1988). Algorithmic skeletons: a structured approach to the management of parallel computation. Research Monograph on Parallel and Distributed Computing.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association.

do Nascimento, L. T., Ferreira, R. A., and Guedes, D. (2005). Scheduling data flow applications using linear programming. In *ICPP*, pages 638–645. IEEE.

Ferreira, R., Meira, W., Guedes, D., Drummond, L., Coutinho, B., Teodoro, G., Tavares, T., Araújo, R., and Ferreira, G. (2005). Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD*, pages 159–167. IEEE.

Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395.

Gibbons, J. (1996). The third homomorphism theorem. *J. Funct. Program.*, 6(4):657–665.

Góes, L. F. W., Guerra, P. H. C., Coutinho, B., da Rocha, L. C., Jr., W. M., Ferreira, R., Neto, D. O. G., and Cirne, W. (2005). Anthillsched: A scheduling strategy for irregular and iterative i/o-intensive parallel jobs. In *JSSPP*, pages 108–122. Springer.

Gupta, R. and Amarasinghe, S. P., editors (2008). *Proceedings of the PLDI*. ACM.

Hall, M., Padua, D., and Pingali, K. (2009). Compiler research: the next 50 years. *Commun. ACM*, 52(2):60–67.

Hofstee, H. P. (2005). Power efficient processor architecture and the cell processor. In *HPCA*, pages 258–262. IEEE.

Johnson, T. A., Lee, S. I., Fei, L., Basumallik, A., Upadhyaya, G., Eigenmann, R., and Midkiff, S. P. (2004). Experiences in using cetus for source-to-source transformations. In *LCPC*, pages 1–14. Springer.

Kongetira, P., Aingaran, K., and Olukotun, K. (2005). Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29.

Macqueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *Procedings of the Fifth Berkeley Symposium on Math, Statistics, and Probability*, volume 1, pages 281–297. University of California Press.

Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M. (2007). Automatic inversion generates divide-and-conquer parallel programs. In *PLDI*, pages 146–155. ACM.

Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated- *ll(k)* parser generator. *Software, Practice and Experience*, 25(7):789–810.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and mei W. Hwu, W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP*, pages 73–82. ACM.

Scaife, N., Horiguchi, S., Michaelson, G., and Bristow, P. (2005). A parallel sml compiler based on algorithmic skeletons. *J. Funct. Program.*, 15(4).

Spring, J. H., Privat, J., Guerraoui, R., and Vitek, J. (2007). Streamflex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228. ACM.

Steinhaus, H. (1956). Sur la division des corp materiels en parties. *Bull. Acad. Polon. Sci*, 1:801–804.

Witten, I. H. and Frank, E. (2002). Data mining: practical machine learning tools and techniques with Java implementations. *ACM SIGMOD Record*, 31:76–77.