Use Parallel Backtracking Algorithm to solve matching problem

Computer Algebra Research Group
of
Wilfrid Laurier University

by Chunxiao (Ian) Li

Supervised by Professor Ilias Kotsireas, Physics & Computer Science, WLU

March 9th, 2016

Problem Description

Given n files (usually n = 2, 3 or 4) of 2D matrices of integers where all of the files have k columns, and a target number lambda λ.

We want to find combinations of n rows (1 row from each file) where

$$F^1_{r_1,c_1} + F^2_{r_2,c_1} + \ldots + F^n_{r_n,c_1} = \lambda$$
$$F^1_{r_1,c_2} + F^2_{r_2,c_2} + \ldots + F^n_{r_n,c_2} = \lambda$$
$$\ldots$$
$$F^1_{r_1,c_k} + F^2_{r_2,c_k} + \ldots + F^n_{r_n,c_k} = \lambda$$

We denote a possible solution by $[r_1, r_2, \ldots, r_n]$, where $r_1, r_2, \ldots, r_n$ are the line numbers, and such solution is called a **matching**.

1 Approaches

1.1 Brute force approach

A straight forward brute force approach will be generating all combinations of rows and verify if any of the combinations are valid. However this is really inefficient where for four files with a, b, c, d rows respectively and k columns, there are $a*b*c*d$ combination of rows and each combination will take $3k$ summations to verify, where the run time will be $O(3k(a*b*c*d))$, this doesn't seem too bad until when a, b, c, d are 1 million lines each!

1.2 Backtracking approach

Generally speaking backtracking is Depth First Search(DFS) with tree pruning technique. Where we incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

1. **Pruning in n-way matching:**

   At column one (root level, $currentDepth = 0$) we find all combinations of rows that have first column numbers adding up to λ, each combination will form a tree branch from the root, at each iteration we choose one of the branches from current level and proceed to the next column only using rows involved in that combination and temporarily ignoring invalid ones.

   On top of this we can still do better, observing the nature of the files, although there can be millions of rows in a file, number of different numbers are only around 10-30, we can then take advantage of this property using BIN method.

2. **BIN:**

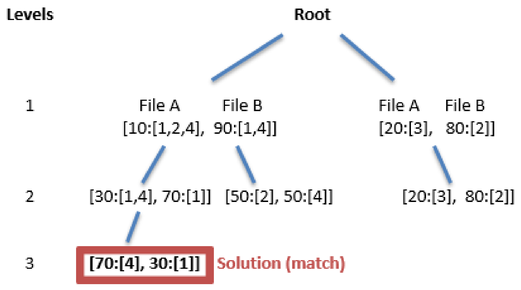
   The idea of BIN is a preprocessing that groups all the rows that have same number at $column = currentDepth$ of the search tree together. We have now translated the problem from "finding combination of rows" to "finding combination of numbers". And this avoids doing repetitive calculation when few thousand lines have the same number at a certain column.

A:

10	30	40
10	50	60
20	20	50
10	30	70

B:

90	70	30
80	80	10
20	80	90
90	50	60



Above is an example of an instance of the problem with two input files and $\lambda = 100$.

$[10 : [1, 2, 4], 90 : [1, 4]]$ at level 1 means:

$A_{1,1}, A_{2,1}, A_{4,1} = 10$
$B_{1,1}, B_{4,1} = 90$.

and

$[1, 1], [1, 4], [2, 1], [2, 4], [4, 1], [4, 4]$ are possible solutions(matches).

2 Implementation

2.1 An approach using C++ Template

A backtrack implementation was introduced [1] which uses templates that allow the program to use any STL container to store the decision tree. The template arguments allow you to specify your data type (int, enum, etc.), the container that stores the decision tree (vector, c array, etc.), and the user-defined function that evaluates the decision tree's correctness.

This approach is great however it won't fit for our case, the implementation with templates described in the paper has one limitation that the domain size of each level must be same, which does not hold in our case, because each time we visit a level, we have distinct sets of rows. However this approach inspired me in understanding the nature of backtracking.

3 Serial vs Parallel

3.1 Serial Farming:

This backtracking approach can be simply implemented on a serial program that runs on one processor by traversing the search tree in DFS manner. However as we traverse along one of the branches, we can also traverse along another without two of them interfering each other, and thus we advance the serial program to a parallel version using multiple processors. (ranging from few processors to hundreds of them)

3.2 Parallel version with MPI:

Naive parallel version:

The naive version works as following:

1. We have a Dispatcher process that dispatches jobs to Worker processes.
2. Dispatcher process has a list of idle Workers that are waiting for jobs.
3. Dispatcher process starts by BIN-ing the first column and initializing the root level of the search tree.
4. Dispatcher sends one of the combinations (branch) to an idle Worker.
5. Dispatcher keeps listening to Worker process for completion signal.
6. Dispatcher adds the completed Worker process to idle list.
7. Repeat until the root is empty and idle list is full again.

Problems with the naive version:

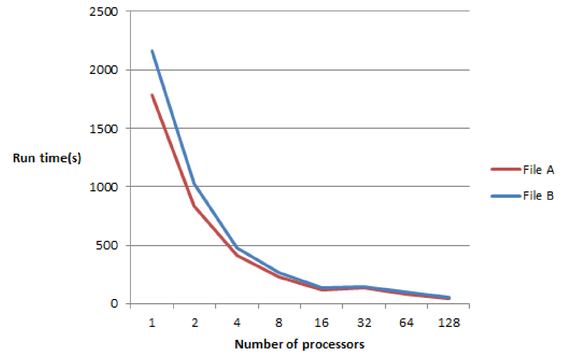
1. When number of branches from root is smaller than the number of Workers given, since we are giving one branch to a Worker, there will be Workers with no jobs to execute and have to wait till the end of the program terminates, another way of saying this is we are wasting lot of resources.
2. Due to the nature of the input files, the numbers of branches are Gaussian distributed both globally and locally. And this will cause the same problem as the first one, after root is empty, there will eventually be Workers idling and others busying, and in practice the there can be hundreds of processors waiting for days!

Parallel with load balancing:

The load-balancing version works as following:

1. We have a Dispatcher process that dispatches jobs to Worker processes.
2. Dispatcher process has a list of idle Workers that are waiting for jobs.
3. Dispatcher process starts by BIN-ing the first column and initializing the root level of the search tree.
4. Dispatcher sends one of the combinations (branch) to an idle Worker.
5. Dispatcher keeps listening to Worker process for completion signal.
6. Dispatcher adds the completed Worker process to idle list.
7. Repeat until the root is empty.
8. while idle list is not full, Dispatcher randomly associate a busy process with an idle process, and the idle process will send a request to the busy one, if the busy one still have some branches left, it donate half of its top level branch to the idle one.
9. repeat until idle list is full again.

4 Performance



Above figure describes the relation between number of processors used and time the program took to exhaustively search two different sets of files. This proves the algorithm is efficient.

References

[1] Roger Labbe. *Solving Combinatorial Problems with STL and Backtracking*.
http://www.drdobbs.com/cpp/solving-combinatorial-problems-with-stl/184401194

This poster has been realised with LaTeX using Lars Nummendal A0 poster template