Parallel and Distributed Algorithms and Programs (APPD) Programming Project

Oguz Kaya
oguz.kaya@ens-lyon.fr

Pierre Pradic
pierre.pradic@ens-lyon.fr

14/11/2016

All source files and documents are available on `http://www.oguzkaya.com/teaching`
DEADLINE: *Sunday, December 18, 23h59 (Paris time)*

Label propagation is a very simple yet a very popular and effective method in solving many types of graph problems. Given a graph $G = (V, E)$ with the set V of vertices and the set E of edges, label propagation aims to assign a label $l(v)$ to each vertex v where $0 \leq l(v) < K$, and K is the number of available labels. It is an iterative algorithm; it starts with an initial label assignments on vertices (typically random), then updates the labels of each vertex $v \in V$ at each iteration, according to the labels of the neighbors of v in the previous iteration. In our case, we will be exclusively dealing with the case where $K = 2$, and assigning the labels 0 and 1 to vertices.

For a given graph $G = (V, E)$, and an imbalance tolerance $\epsilon \geq 1.0$, the graph bisection problem is defined as partitioning the vertex set V into two disjoint sets V_1 and V_2 where $V = V_1 \cup V_2$ in a way that the partition is balanced, i.e., $|V_1|, |V_2| \leq (|V|/2)\epsilon$, and the $cutsize(E, V_1, V_2)$ of the partition is minimized. $cutsize(E, V_1, V_2)$ is defined as the number of edges in E connecting parts V_1 and V_2 (specifically, the number of edges whose one end is connected to a vertex in V_1, and the other end connected to a vertex in V_2, or vice versa). In Figure 1, there is a balanced bipartition for $\epsilon = 1.0$ with cutsize 7 (there are exactly 7 edges connecting these two parts) for a graph having 10 vertices.

Our goal in this project is find these two vertex sets using a binary label propagation algorithm (i.e., vertices labeled 0 will constitute V_0, those with label 1 will form V_1). The pseudocode for this algorithm is given in Algorithm 1. We start by assigning each vertex to a part randomly. Then, for each vertex, we first check if we are allowed to move the vertex to the other part (i.e., other part has less than $(|V|/2)\epsilon$ vertices). If so, we move the vertex to the other part, if the vertex has more neighbors in that part. Otherwise, the vertex stays in the original part. We iterate the algorithm for *maxIters* iterations, and print the cutsize and the load imbalance at each iteration.

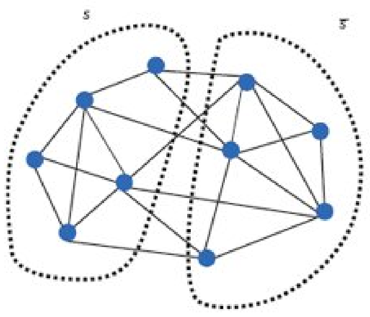
To create a random graph, we provide a script named `create-graph.py` that takes the number of vertices $|V|$ in the graph as the first argument, the number of edges in the graph as the second argument, and the name of the output file for the graph as the third argument. Now, try to create a sample graph by typing

```
./create-graph.py 10 40 graph.txt.
```

This will create a graph with 10 vertices and 40 edges, and write it in the file `graph.txt`.

As usual, we provide a skeleton code `graph-bisect-skeleton.c` that you should not touch, and a solution file `graph-bisect-solution.c` that you are expected to fill out with your solutions.

Figure 1: A 10-vertex graph bisected with a perfect load balance ($\epsilon = 1.0$) and the cutsize 7. The cluster on the left corespond to vertices having label 0, wheres the vertices one on right side are labelled 1.



Algorithm 1 Label propagation-based graph bisection

```
Input: $G = (V, E)$: Graph to be partitioned
Output: $part$: A 0/1 array of size $|V|$ indicating to which part each vertex belongs to.
  $part \leftarrow$ RandomBinaryArray($|V|$)          ▶ Initialize each of $|V|$ elements to 0 or 1 randomly.
  PrintCutSizeImbal($G, part$)          ▶ Print the cutsize and the imbalance of the partition.
  for $i = 1 \dots maxIters$ do          ▶ Label-propagation iterations
    for $j = 1 \dots |V|$ do          ▶ Iterate over each vertex $v_j$, and update its nextPart based on its neighbors.
      $otherPart = 1 - part[j]$          ▶ The other part, which does not involve the vertex j
      if otherPart has more than $(|V|/2)\epsilon$ vertices then          ▶ otherPart is overloaded; vertex j cannot move.
        $nextPart[j] \leftarrow part[j]$
      else
        if vertex j has more neighbors in otherPart then          ▶ Neighbors in the otherPart propagate their part
          $nextPart[j] \leftarrow otherPart$
        else          ▶ The otherPart is overloaded; vertex $j$ cannot move.
          $nextPart[j] \leftarrow part[j]$
    $part \leftarrow nextPart$          ▶ Update the part array with the new parts.
    PrintCutSizeImbal($G, part$)          ▶ Print the cutsize and the imbalance of the partition.
  return $part$
```

Open up the file `graph-bisect-solution.c`. There is a skeleton implementation of a function with the signature

`bisectGraph(int N, int *adjBeg, int *adj, int *part, int maxIters, double epsilon, char *algoName).`

The arguments of the function corresponds to the following:

- **N**: The number of vertices in the graph.
- **adj**: The list including the the indices of adjacent vertices.
- **adjBeg**: The pointer indicating the beginning of the list of adjacent vertices for each vertex. These pointers are consecutive; the beginning pointer for the vertex v_{j+1} is the end pointer for the vertex v_j. For example, for the vertex v_j, its adjacent vertices are those with indices $adj[adjBeg[j]], adj[adjBeg[j]+1], \dots, adj[adjBeg[j+1]-1]$. Similarly, the list of neighbors for the vertex v_{j+1} contains vertices with index $adj[adjBeg[j+1]], adj[adjBeg[j+1]+1], \dots, adj[adjBeg[j+2]-1]$, and so on.
- **part**: The array to be filled with the part indices of each vertex (0 or 1). You can assume that the array is already allocated and initialized randomly with 0s and 1s, and that each MPI process has the same copy of the array.
- **maxIters**: The number of iterations for which algorithm should run.
- **epsilon**: Maximum allowed imbalance parameter, `epsilon` ≥ 1.0.
- **algoName**: The name of the algorithm to be executed.

You can assume that each MPI rank has the same initial copy of the graph data (`N, adjBeg, adj`) and the `part` array.

The function is divided into 3 sections, each corresponding to a specific algorithm name. For the algorithm name **bisect-seq**, you will implement a sequential version of the label propagation-based graph bisection given in Algorithm 1. For **bisect-a2a**, you will implement parallel bisection using all-to-all communication routines (MPI_Allgatherv). For **bisect-p2p**, you will implement parallel bisection using point-to-point communication (MPI_Send and MPI_Recv). In any case, the goal of the algorithm is to compute the part index for each vertex $v_j \in V$ using the label propagation, and put this into the entry $part[j]$. In any case, you should make sure that the final `part` array resides at the process with rank 0.

We also provide the following function

`printCutSizeImbal(int N, int *adjBeg, int *adj, int *part)`

that prints the cutsize and the load imbalance of the partition provided in **part**. Make sure to call this function at the end of each iteration. We will use these two values to test the correctness of your code. Do not print anything else in the final code that you submit.

Part 1

Graph Bisection with Label Propagation (GBLP) (20 points)

Question 1

a) Implement the label propagation-based graph bisection algorithm sequentially, using the algorithm name **bisect-seq**. Make sure that the algorithm runs for `maxIters` iterations, just as given in Algorithm 1. Use the given function **printCutSizeImbal** to print the cutsize and the imbalance of a partition at the end of each iteration.

Part 2

Parallel GBLP with All-to-All Routines (40 points)

Question 2

a) Implement a parallel version of the previous label propagation algorithm. In the parallel version, each process (except the last) will update the labels of $\lceil N/P \rceil$ vertices with consecutive indices. For instance, the process 0 will update the labels of the vertices $v_0, \dots, v_{\lceil N/P \rceil - 1}$, the process 1 will update the labels of $v_{\lceil N/P \rceil}, \dots, v_{2\lceil N/P \rceil - 1}$, and so on (except the last process, who will update for the last $(N - (N-1)\lceil N/P \rceil)$ vertices). Use the given function **printCutSizeImbal** to print the cutsize and the imbalance of a partition at the end of each iteration.

b) Now that each process updates a portion of size $\lceil N/P \rceil$ of the `part` array at the end of an iteration, we need to communicate all these subarrays so that each process has the new value of the whole `part` array. Perform this using the routine MPI_Allgatherv (as we saw in the TP session), so that each process makes available the $\lceil N/P \rceil$ elements of the `part` array that it updates locally.

Part 3

Distributed Parallel GPLP with Point-to-Point Communication (40 points)

Question 3

a) In this question, we will use the exact same parallel implementation as the previous question to update the part arrays locally. However, we will change the communication step for the `part` array, and optimize this step. Remember, for instance, that the process 0 updates the vertices $v_0, \dots, v_{\lceil N/P \rceil - 1}$. In doing this update, we generally do not need the part values of all vertices. Instead, the part values of the neighbors of the vertices $v_0, \dots, v_{\lceil N/P \rceil - 1}$ are sufficient (see Algorithm 1). At each process, we will therefore identify this set of neighboring vertices at each process so that a process will communicate the part value of a vertex v_j it owns solely to processes owning some neighbor of v_j. Note that the index of the owner process of the vertex v_j is $\left\lfloor \frac{j}{\lceil N/P \rceil} \right\rfloor$, as our partitioning is contiguous in the vertex indices. Before we perform the label propagation iterations, we need to compute the following communication structs that will enable point-to-point communication afterwards. You are free to follow these instructions step-by-step, or to deviate it to do it your own way, as you see fit. But in any case, the communication should be "optimal" in the sense that the value `part[j]` should only be sent to processes that own a neighbor vertex of v_j; otherwise, you will not get full credit.

- At each process, make a pass over the adjacency lists of its vertices. Allocate an array `recvCount` of size P, and for each `recvCount`[i], compute the number of unique adjacent vertices belonging to the process i. `recvCount`[i] indicates the number of vertices in another process i whose `part` values need to be sent to the current process.
- Now allocate a pointer array `recvIdx` of size P, and for each pointer `recvIdx[i]` allocate an `int` array of size `recvCount`[i]. Fill in each list `recvCount`[i] with the indices of these adjacent vertices whose `part` values are to be received from the process i.
- Now that each process knows how many elements it receives from all others, we will compute another array with elements `sendCount[i]` indicating how many elements need to be sent to each process i. Node that if the current process receives `recvCount[i]` elements from the process i, then process i needs to send this many elements to the current process (which defines its corresponding `sendCount` value). Do the communications, using MPI_Alltoall or MPI_Isend/Recv, and compute these arrays.
- After computing the `sendCount` array, similarly create a pointer array `sendIdx` of size P, and allocate an array of size `sendCount[i]` for each pointer `sendIdx[i]`. Note that a process receives the `part` values of the vertices whose indices are in `recvIdx[i]` from the process i. Therefore, the process i should have the same indices in its corresponding `sendIdx` list. Perform this communication using MPI_Isend and MPI_Recv, or MPI_Alltoallv.

- Create two pointer arrays `recvPart` and `sendPart` of sizes P, and allocate `int` arrays of size `recvSize[i]` and `sendSize[i]` for each `recvPart[i]` and `sendPart[i]`, for $i = 1, \ldots, P$. We will use these arrays as buffers in the point-to-point communication.
- Now, copy-paste the code you implemented in the previous question for the label propagation loop. Remove the communication part which uses MPI_Allgatherv. For each $i = 1, \ldots, P$, fill in the array `sendPart`[i] using the locally updated values of `part`, send it to the process i using MPI_Isend, and similarly receive the updated part values at process i into `recvPart`[i] using MPI_Recv, finally scatter these values onto the `part` array using the corresponding indices `recvIdx`[i]. Make sure you post all MPI_Isends to all processes first, then start posting MPI_Recvs, to avoid deadlocks. In this way, only the necessary elements of the `part` array would be communicated.

You are encouraged to use Simgrid/SMPI to benchmark and optimize your code. In using SMPI to compile and run your code, you need to avoid using C99 constructs, global variables, and complicated macros, to avoid errors.

Guidelines

- Source files for the skeleton codes are available at `www.oguzkaya.com/teaching`. In case we update or change something, we will put the updated files to the website, and inform you of this update by e-mail.
- You need to use the C language and the MPI library for programming. Do not use C99 features (such as for(int i = 0...)). Do not use any fancy non-standard C libraries that do not exist in all Linux/Unix distros; code these functions by yourself. Do not use any other language, scripts, whatever.
- Normally, you should not need any additional source files. If you like to have more files anyways, make sure that you modify the Makefile accordingly. Your submission must include all the source files (*.c), header files if exists (*.h), I will be using mpicc as the compiler, and running your code on a real parallel machine for benchmark. In any case, never ever try to modify `graph-bisect-skeleton.c`.
- Make sure that the Makefile that you provide compiles the code using mpicc on any of the lab computers (Salle E001). If the code does not compile with the Makefile, you automatically get 0 points.
- Do not print anything else in the code, except the output of `printCutSizeImbal` at the end of each iteration. We will use this output to test the correctness of your code.
- Send your submission as a single tar file with the format `[surname]-[name].tar.gz` to `oguz.kaya@ens-lyon.fr`. In the subject line, put "APPD PROJECT [SURNAME] [NAME]". If you ever need to resubmit due to a mistake you correct in the last minute, do it by simple versioning. For instance, for the second submission, put "APPD PROJECT [SURNAME] [NAME] V2" in the subject, etc. Try to avoid multiple submissions as much as possible. We will use only the latest version of each submission.

Grading Policy

Our expectation is that everyone does the first question without any problems (which involves no parallelism), and most of you implement the second question correctly (as this is similar to the TP session). The third question takes some thinking and implementation effort; but this is very well documented, and we have no doubt that many among of you will manage to nail it!

Your codes will be evaluated according to the following criteria:

- Accuracy (40%): Whether your program does what is asked in the question. If the program has bugs, produces incorrect answers, etc, you will lose points depending on how bad it is.
- Speed (35%): How fast your code runs, in compare to other people's code (A little competition does not hurt). Let's get those optimizations going on! Note that if you have nice optimization ideas implemented, even if they do not yield very good speedups, you might get points if they are well documented for us to understand.
- Code Quality (25%): How good and neatly your code is written in general. Consistency of your indentations, variable and procedure naming, the structure of your code (you don't want to implement everything in one function having 1000 lines!), and of course, most importantly, how well your code is commented for us to understand (what does each variable, function, code segment do?) Anyhow, do not expect to get partial credits by neatly writing some irrelevant code to the questions you could not solve.

Late policy: For each hour of late submission after the deadline, you lose 1 point (out of 100).