Massachusetts Institute of Technology — Handout 9
6.892: Theory of Parallel Systems — February 13, 1997
Professor Charles E. Leiserson

# Lab 1 — Data-parallel lab

This assignment is due in class on *Thursday, February 27.* This is a long assignment, and therefore plan your time accordingly. Your final program will likely be short (say, less than 5Kb), but it will take you a while to write it.

In this assignment you will implement the **preflow-push** algorithm for the maximum flow problem. The algorithm is described in detail in Section 27.4 of CLR. You will program in NESL; the NESL language is described in a separate handout.

Turn in your solution on paper, including a printout of the source code and some test cases. Moreover, you should indicate the location of your code on `xolas`, so that we can run your program and check that it works.

## 1 Maximum Flow

In this section we shall define the maximum flow problem and introduce the notation used in the rest of the paper (this is just a quick review of CLR, Section 27.1).

A **flow network** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative **capacity** $c(u, v) \geq 0$. If $(u, v) \notin E$, we assume that $c(u, v) = 0$. We distinguish two vertices in a flow network: a **source** $s$ and a **sink** $t$. For convenience, we assume that every vertex lies on some path from the source to the sink.

A **flow** in a flow network $G$ is a real-valued function $f : V \times V \mapsto \mathbf{R}$ that satisfies the following three properties:

**Capacity constraint:** For all $u, v \in V$, we require $f(u, v) \leq c(u, v)$.

**Skew symmetry:** For all $u, v \in V$, we require $f(u, v) = -f(v, u)$.

**Flow conservation:** For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(u, v) = 0 \ .$$

The **value** $|f|$ of a flow $f$ is the net flow into the sink:

$$|f| = \sum_{v \in V} f(v, t) \ .$$

A **maximum flow** is a flow of maximum value.

The problem we wish to solve is that of computing a maximum flow in a given network. We shall use Goldberg's **preflow-push** algorithm, which is one of the fastest known algorithms for the problem, and is not too difficult to parallelize.

*At this point, you* **must** *read carefully Section 27.4 of CLR.*

# 2 Parallelization of Goldberg's algorithm

In this section we show a parallel variation of Goldberg's algorithm for maximum flow.

Recall the code for GENERIC-PREFLOW-PUSH on page 609 of CLR:

```
GENERIC-PREFLOW-PUSH(G)
INITIALIZE-PREFLOW(G, s)
while there exists an applicable push or lift operation
      do select an applicable push or lift operation and perform it
```

The generic algorithm does not specify any particular order in which PUSH and LIFT operations should be applied, and there are indeed many variations on Goldberg's algorithm.[1] We are thus free to chose any order that allows us to expose parallelism. We shall not explain the parallel algorithm in detail, but here is the basic idea. The parallel algorithm proceeds in **pulses**, each of which consists of a number of operations applied in parallel. Each pulse is divided into two stages. The first stage pushes excess flow out of nodes: it applies the 'push' operation to all nodes to which it is applicable, in parallel. The second stage lifts (in parallel) all the nodes that can be lifted.

**(a)** Prove that, on input $G = (V, E)$, the parallel algorithm terminates after at most $O(|V|^2)$ pulses.

# 3 Input and Output

In this section we describe the format of the input graphs your program should accept, and the output format that the program should produce.

The input graph is given as a triple $X = (E, \text{source}, \text{sink})$, where $E$ is a sequence of edges, source and sink are respectively the source and the sink. Each edge has the form $((u, v), c)$, where the edge goes from node $u$ to node $v$, and has capacity $c$. The sequence of edges is given in no particular order. Notice that no explicit list of vertices is given: you can assume that the vertices are labeled with the integers from 0 to $|V| - 1$. Therefore, source, sink, $u$ and $v$ are integers in the range $[0 \ldots |V| - 1]$. The capacities $c$ are floating-point numbers. For example, here is a representation of the graph in Figure 27.6 of CLR:

```
X =  ([((0, 1), 16.0), ((0, 3), 13.0), ((1, 2), 12.0),
       ((1, 3), 10.0), ((2, 3), 9.0), ((2, 5), 20.0),
       ((3, 1), 4.0), ((3, 4), 14.0), ((4, 5), 4.0), ((4, 2), 7.0)],
      0, 5)
```

[1] See also J. S. Bach, *Goldberg Variations*, BWV 988, for an earlier independent set of variations on the same subject.

As for the output format, your program must produce a triple $Y = (F, e_{\text{source}}, e_{\text{sink}})$, where $F$ is a sequence that contains the flow of all edges, and $e_{\text{source}}$ and $e_{\text{sink}}$ are respectively the excess of the source and the sink. If $e_{\text{source}} \neq -e_{\text{sink}}$, then your program is buggy. Each element of $F$ has the form $((u, v), f)$, where $f$ is the flow on the edge $(u, v)$. The elements of $F$ can be in any order. All edges whose flow is nonzero must be part of the output, but feel free to also include edges whose flow is zero if you find it convenient.

Here is an example of what your program is expected to produce:

```
<Nesl> MaxFlow(X);

Compiling..Writing..Loading..Running..
Exiting..Reading..
it =
  ([((0, 1), 16.0), ((0, 3), 7.0), ((1, 0), -16.0), ((1, 2), 12.0),
    ((1, 3), 4.0), ((2, 1), -12.0), ((2, 3), 0.0), ((2, 4), -7.0),
    ((2, 5), 19.0), ((3, 0), -7.0), ((3, 1), -4.0), ((3, 2), 0.0),
    ((3, 4), 11.0), ((4, 2), 7.0), ((4, 3), -11.0), ((4, 5), 4.0),
    ((5, 2), -19.0), ((5, 4), -4.0)], -23.0, 23.0)
   : [((int, int), float)], float, float
```

# 4 Internal representation

In this section we describe the representation of the graph that your implementation should use internally. Such a representation is different from the input format, and allows for an easy implementation of the preflow-push algorithm (as well as many other graph problems).

We shall describe the internal representation with an example. Consider a graph with 3 nodes $A$, $B$ and $C$, and 3 edges $(A, B)$, $(B, C)$ and $(C, B)$. First, we want to ensure that, for every edge $(u, v)$, the opposite edge $(v, u)$ also belongs to the graph (we can always add an edge of zero capacity to the input graph, without altering the solution of the maximum flow problem). This means that we must add one more 'dummy' edge $(B, A)$. Second, we sort the list of edges lexicographically, and we get the sequence $[(A, B), (B, A), (B, C), (C, B)]$. From now on, this will be our canonical order of edges. The next step is to build two auxiliary sequences. We first build a list containing the **out-degree** of all vertices. In the example, the out-degree sequence will be $[1, 2, 1]$. The sequence of degrees is useful because in NESL we can `partition` the sequence of edges using the list of the degrees, and get the adjacency list of the graph: $[[(A, B)], [(B, A), (B, C)], [(C, B)]]$. The second auxiliary sequence we need is the list of **mates**. For an edge $(u, v)$ in the $i$-th position of the sequence of edges, mate$[i]$ will give the position of the edge $(v, u)$ in the same sequence. Clearly, mate[mate$[i]$] $= i$ for all $i$. Recalling that in NESL indices start at 0, we get a sequence of mates $[1, 0, 3, 2]$.

For ease of implementation of the subsequent phases, it is more convenient to store the two elements of the pairs that represent edges in separate sequences, that we call `src` and `dst`. In the example, `src` is $[A, B, B, C]$. In the same way, keep a separate sequence for every value associated with an edge — in particular, the capacity of the edge.

**(b)** Write a NESL function `ConvertInput` that transforms the input format into a 7-tuple `G = (degree, src, dst, mate, c, source, sink)`, where the first four components are as described above, `c` is the sequence of capacities, and `source` and `sink` are as in the input format.

You will now master the internal representation with a couple of warm-up exercises. You will write a function `MaxDegreeNeighbor(G)` that returns a sequence of integers $m_i$, one per vertex. The integer $m_i$ is the maximum of the degrees of all the neighbors of the $i$-th node.

There are at least two ways to write `MaxDegreeNeighbor` in our representation. The following function uses concurrent reads (the `->` operator):

```
function MaxDegreeNeighbor(G) =
let
    (degree, src, dst, mate, c, source, sink) = G;
in
    { max_val(nd) : nd in partition(degree->dst, degree) }
$
```

**(c)** Write `MaxDegreeNeighbor(G)` *without* using concurrent reads from the sequence of degrees.

# 5 Implementation of the preflow-push algorithm

You are now ready for the main part of this assignment.

**(d)** Write a function `MaxFlow(X)` that takes a graph `X` (in the input format), converts it into internal format, computes the maximum flow, and returns it in the format described in Section 3. *Hint:* Do not worry about the output format until you are done.

**(e)** **(Extra credit)** Suggest improvements to the algorithm. Improve the implementation. Try other internal representations. Be creative.

# A Using NESL on `xolas`

In this appendix we describe how to edit and run NESL programs on `xolas.lcs.mit.edu`. There are many `xolas` machines. Please use only `xolas0` and `xolas1`.

We set up a directory on `xolas` that contains stuff that you may find useful for completing the assignment. The directory is

```
/usr/local/6.892/lab-1
```

We shall refer to this directory as the **lab** directory. Moreover, you can look at the NESL examples in the NESL directory,

```
/usr/local/nesl/examples
```

There are more examples on the NESL web page at CMU; follow the link from the class web page if you want to get there.

The lab directory contains some examples (`.in` files). For each example, the corresponding `.sol` file contains the value of the maximum flow.[2] There is also a file `read-input.nesl` that contains a function to read an input file into NESL. Try

```
X = ReadInput("CLR-27.6.in");
```

to read the example in Figure 27.6 of CLR.

The biggest example is `m50.in`. As a matter of comparison, our program takes 150 seconds to solve the problem on this graph. Our program is pretty naïve, and you can certainly do much better than that.

The best way to use NESL is from within Emacs. Copy the file `.emacs` from the lab directory into your home directory. This suffices to be able to run NESL, as described in Section 4 of the NESL user manual. Just type `M-x run-nesl` and start hacking:

```
GCL (GNU Common Lisp)  Version(2.2) Fri Feb  7 13:00:02 EST 1997
Licensed under GNU Public Library License
Contains Enhancements by W. Schelter

% NESL version 3.1.0 (Wed Dec 20 14:26:22 EST 1995) %

% Setting machine configuration LOCAL.... %

% Use (nesl) when an error aborts you out of the interpreter. %
% Type help; for a list of the top level commands. %

<Nesl> 1 + 1;

Compiling..Writing..Loading..Running..
Exiting..Reading..
it = 2 : int
```

Or, less naïvely, you can use parallel prefix to print "Hello, world!":

[2] The solution has been computed by two different programs, written in NESL and C. We are pretty sure that the solution is correct.

```
<Nesl> magic = [72, 29, 7, 0, 3, -67, -12, 87, -8, 3, -6, -8, -67];

Compiling..Writing..Loading..Running..
Exiting..Reading..
magic = [72, 29, 7, 0, 3, -67, -12, 87, -8, 3, -6, -8, -67] : [int]
<Nesl> { code_char(q+magic) : q in plus_scan(magic); magic };

Compiling..Writing..Loading..Running..
Exiting..Reading..
it = "Hello, world!" : [char]
```

You are actually interacting with a Common Lisp system (the NESL compiler is written in LISP). Sometimes a Lisp error occurs, and the system goes back to the Common Lisp command loop. In this case, as the help message suggests, type `(nesl)` to re-enter NESL.

Such an error occurs systematically when the result of your computation is a huge array:

```
<Nesl> [1:10000];
Compiling..Writing..Loading..Running..
Exiting..Reading..
Warning: Result is large, it will take a while to read.
Error: Value stack overflow.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by FUNCALL.
Broken at SYSTEM:TOP-LEVEL.  Type :H for Help.
NESL-LISP>>
```

After a couple of these errors, the Common Lisp interpreter crashes horribly. The best way to avoid the problem is by using **file variables**. File variables are described on page 8 of the user manual.

NESL has some debugging and profiling support (see Section 3.4 of the manual). We found the `set trace` command particularly useful.

# B Hints, tips and tricks

- Like many other programming languages, NESL has a list of reserved words. NESL is particularly unfortunate in this respect. `f` (your favorite variable name for the flow function) and `t` (your favorite variable name for the sink) are the predefined boolean values, and you cannot use them. Also, NESL is case-insensitive (as opposed to the rest of the glorious Unix tradition): `f` and `F` are the same thing, and you can use neither.

- In NESL, integers and floating-point numbers are not compatible. NESL will not convert 1 into 1.0: you need to explicitly convert it. In this assignments, all capacities are floating-point numbers (and, consequently, the flow and excess are floating-point quantities, too).

- Beware that the terminator character for function definitions is a dollar sign `$`, and not a semicolon `;`.
- NESL may have bugs. We tried the CM-5 version which was definitely buggy (this is why you are working on `xolas` instead). If things do not work as you expect, consider the possibility that there be a bug in NESL.
- The Common Lisp system may have bugs, too. There is one bug we know of: the Lisp interpreter crashes if you use the tilde character ~ in file names. We fixed the bug for the special case `~/path`, but `~user/path` will not work.[3]
- Do not look for NESL on the web. You will find billions of links to the New England School of Law. There is a link to NESL on the class web page.
- The network flow problem was the topic of the First DIMACS Implementation Challenge (1990–1991). If you are curious, look at `ftp://dimacs.rutgers.edu/pub/netflow/`. *Hint:* There is no parallel code there, anyway.
- If you are in trouble, do not despair, and send email to the friendly TA.

[3] This is actually a bug in Solaris, not in the GNU Common Lisp. Solaris traverses an unbelievable number of software layers even for a simple operation like "read `/etc/passwd`".