TR-533

Lin-Kernighan partitioning algorithm on Multi-PSI

by
D. Dure

February, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Lin-Kernighan partitioning algorithm on Multi-PSI

12/21/89

Daniel Dure

Abstract:

Partitioning a graph into several sets while keeping the connection cost between sets low can be useful to minimize communication cost in distributed algorithms, or to provide an initial placement to more sophisticated algorithms. If there are n nodes in the graph, and if partition holds s sets, the complexity of the sequential Lin-Kernighan algorithm is $\mathcal{O}(n^2 \log n)$. Our parallel implementation nearly divide this figure by $s/2$, provided that this number of processors are available. Performance analysis is provided for a number of example graphs.

Institute for New Generation Computer Technology

Fourth Laboratory

Forewords

The following document is intended of course as the presentation of an useful program for subsequent applications, notably in VLSI CAD. It is also an example of KL1 programming on the Multi-PSI, while using the FLIB library. Although there are certainly many mischiefs to pick up here ot there, we dare believe these few lines can be read by a novice KL1 programmer with the hope of getting some useful hints.

Acknowledgements

We would like to thank some people who helped us, either during the course of programming or debugging: Kimura Koichi and Yoshida Kaoru demonstrated an endless kindness to lighten the tedium of discovering PSI, Multi-PSI and PDSS. You Inamura quickly relieved me from a nasty KL1 bug. Kazuo Taki provided enlightening pieces of advice.

This work as been sponsored by Inria, Rocquencourt, France through a grant and exchange program with Icot, Tokyo, Japan.

Introduction

The program we describe in this document has for main purpose to build a partition of a graph using a given number of sets, while minimizing the cumulated "cost" of edges going from a vertex in a set to a vertex in a different set. This problem is indeed NP-complete, and we use an heuristic proposed by B.W. Kernigan and S. Lin in [2] to approximate the solution.

This operation is useful, on a large scale, to spread a graph over a plane while keeping the number of long edges as low as possible. As an example, during the design of VLSI circuits, thousands of cells connected by wires have to be spread over a constrained silicon area, and connections between cells, whose length is critical to speed and quality of signal transfer, have to be as short as possible.
Also, when distributing a computation over a network of processors, in order to perform computations in a parallel fashion, it is important to avoid heavy communication between the tasks on different processors, while keeping load balanced between processors. If we represent the computation as a graph, partitioning can help to achieve an efficient distribution.

The following sections describe the KL1 code written to implement the Lin-Kernighan algorithm on a Multi-PSI. Text (and program) organization follows.

- Section 1 describes the module `readfile`, which parses files describing the graph itself, as well as the cost of edges.
- Section 2 describes the module `lk`, which takes the graph and cost information, then compacts and distributes data over processors, after some preprocessing.
- Section 3 describes the module `tree`, which manages communication between processors, distributes the workload and decides to iterate the Lin-Kernighan procedure or to stop operations.
- Section 4 describes the module `algo`, which contains the core of the partitioning algorithm, for two sets.
- Section 5 describes the module use to measure performances.
- Eventually, section 6 holds some examples of graphs and the related performance measurements.

Our reader will notice that modules outside of this list are referred by the following code. They come from the FLIB library that we designed with Bernard Burg. Although predicate names are rather self-explanative, documentation is available in the form of a technical report: [1]. We warmly advice our reader to get hold of this document, especially if he is to program in KL1 on Multi-PSI, circumstances for which FLIB has been tailored.

1 Module readfile

This module contains what is necessary to transform the graph decription data from a file into the vector of strings used by the module `lk`.

```
:- module readfile.
:- public read/3.
```

Structure of the file is extremely simple, and data are in ASCII form. The main purpose of this module is to allow debugging of the Lin-Kernighan algorithm and basic performance measurements.

1.1 File structure

The file holds integers, each one read by a separate `gett`.

1. The first integer is the number of vertices, n.
2. Then come blocks, one per vertex. Each block starts with the vertex number, from 0 to $n-1$. Then, an integers precises the number of connected vertices, v. Then comes the edge information, with two integers for each edge: the label of the connected vertex, between 0 and $n-1$, and the cost of the edge. Cost should be positive of null, if the algorithm is to perform sensible optimization.

On the other hand, the structure used by the module `lk` is a pair of vector of strings, one representing edges starting from each vertex, the other holding the cost associated to each of these edges.

1.2 Reading the file

The following predicate, which is the top level predicate, opens the named file (string or atom) and reads the first integer, in order to create the two vectors which are to hold edge and cost information.

```
read(Name, Edge2, Cost2) :-
    string(Name, _, _) |
        fel:open_file([gett(Vert) | Stream], Name, r, Status),
        (Status = normal, integer(Vert) ->
            new_vector(Edge1, Vert),
            new_vector(Cost1, Vert),
            Stream = [gett(V), gett(E) | Stream1],
            fill(V, E, Stream1, Edge1, Edge2, Cost1, Cost2);
         otherwise;
         Status = normal ->
            util:p_console([string#"file:",Name,string#"has a wrong format"],_),
            Stream = [];
         otherwise;
         true ->
            util:p_console([string#"can't open file:",Name],_)).
read(Name, Edge2, Cost2) :-
    atom(Name) |
        fel:atom_to_name(Name, Name1),
    read(Name1, Edge2, Cost2).
```

The following predicate goes through the file, picking a node number and then parsing the associated edge list, until end of file is met.

```
fill(V, E, Stream, Edge1, Edge3, Cost1, Cost3) :-
    integer(V), integer(E) |
        new_string(Edges, E, 32),
        new_string(Costs, E, 32),
        fillstrings(~(E-1), Edges, Edges1, Costs, Costs1, Stream, Stream1),
        (wait(Costs1), wait(Edges1) ->
            set_vector_element(Edge1, V, _, Edges1, Edge2),
            set_vector_element(Cost1, V, _, Costs1, Cost2),
            Stream1 = [gett(V1), gett(E1) | Stream2],
            fill(V1, E1, Stream2, Edge2, Edge3, Cost2, Cost3)).
otherwise.
fill(end_of_file, end_of_file, Stream, Edge1, Edge2, Cost1, Cost2) :- true |
        Stream = [], Edge1 = Edge2, Cost1 = Cost2.
```

The following predicate parses the cost/edge list.

```
fillstrings(P, Edges, Edges2, Costs, Costs2, Stream, Stream2) :-
    P > -1 |
        Stream = [gett(E), gett(C) | Stream1],
        (integer(E), integer(C) ->
            set_string_element(Edges, P, E, Edges1),
            set_string_element(Costs, P, C, Costs1),
            fillstrings(~(P-1), Edges1, Edges2, Costs1, Costs2,
                        Stream1, Stream2)).
otherwise.
fillstrings(_, Edges, Edges2, Costs, Costs2, Stream, Stream2) :- true |
        Edges = Edges2, Costs = Costs2, Stream = Stream2.
```

2 Module lk

This modules contains the topmost predicates used to perform the lin-Kernighan algorithm, to split a graph into s sets. More precisely, we consolidate the graph and cost information within a single vector of strings, with one string per vertex, and we add dummy vertices, so that the total number of vertices becomes a multiple of the number of sets in the required partition. These artifacts are eventually removed from the partition, after the optimization has been performed.

```
:- module lk.
:- public lk/5.
```

2.1 The topmost (public) predicate

The following predicate operates a min-cut on such a graph, making a s-way partition, where s is specified by `Set`. Operations are done in parallel, using p processors, starting from processor 0, where p is specified by `Proc`. When operations are completed, a string containing the position of each vertex is returned via `Pos`. The string is indexed by the label of each vertex, and the corresponding set is labelled by an integer, ranging from 0 to $s-1$.

The graph itself is specified by vectors `Con` and `Cost`. Each vertex with label l is associated to the string at position l in `Con`. This string is contains the label of vertices connected to l. `Cost` is done the same way, but instead of vertex labels, it contains the cost of the corresponding edge. We consolidate connection and cost matrix to speed up data transfer.

```
lk(Con, Cost, Set, Proc, Pos) :-
    vector(Con, Size), vector(Cost, Size), integer(Set), Proc >= 1 |
        MaxPair := Set / 2,
        fel:mini(Proc, MaxPair, UsedProc),
        init_pos(Con, Con1, Set, Pos0),
        consolidate(Con1, Cost, Con2),
        start_op(Con2, Set, Pos0, UsedProc, Pos1),
        fel:sub_string(Pos1, 0, Size, Pos, _).
```

2.2 Initializing position string and connection matrix

Here, we initialize the position string. Vertices are attributed some position between 0 and $s-1$. We also add "dummy" vertices which are used to balance the size of each set.

```
init_pos(Con, Con1, Set, Pos1) :-
    vector(Con, Size), Rest := Size mod Set, Rest = 0 |
        new_string(Pos, Size, 32),
        new_vector(NewCon, Size),
        init_pos(~(Size-1), ~(Size-1), 0, Set, Con, NewCon, Con1, Pos, Pos1).
otherwise.
init_pos(Con, Con1, Set, Pos1) :-
    vector(Con, Size), Rest := Size mod Set |
        NewSize := Size + Set - Rest,
        new_string(Pos, NewSize, 32),
        new_vector(NewCon, NewSize),
        init_pos(~(NewSize-1), ~(Size-1),
                 0, Set, Con, NewCon, Con1, Pos, Pos1).
```

The following predicate initializes the position strings and connection matrix for the new dummy vertices. Dummy vertices are of course connected to nothing.

```
init_pos(Start, Stop, Modulo, Set, Con, Con1, Con3, Pos, Pos2) :-
    Start > Stop |
        new_string(Void, 0, 32),
        set_vector_element(Con1, Start, _, Void, Con2),
        set_string_element(Pos, Start, ~(Modulo mod Set), Pos1),
        init_pos(~(Start-1), Stop, ~(Modulo+1), Set,
                 Con, Con2, Con3, Pos1, Pos2).
otherwise.
init_pos(_, Stop, Modulo, Set, Con, Con1, Con3, Pos, Pos2) :- true |
        init_pos(Stop, Modulo, Set, Con, Con1, Con3, Pos, Pos2).
```

The following predicate makes the copy for the remaining vertices. Note that may be some naturally "dummy" vertices, i.e. without any reference. In this case, we may use more nodes than necessary. This is unlikely to happen in normal cases, though.

```
init_pos(Stop, Modulo, Set, Con, Con1, Con3, Pos, Pos2) :-
    Stop > -1 |
        set_vector_element(Con, Stop, Old, _, ConC),
        set_vector_element(Con1, Stop, _, Old, Con2),
        set_string_element(Pos, Stop, ~(Modulo mod Set), Pos1),
        init_pos(~(Stop-1), ~(Modulo+1), Set, ConC, Con2, Con3, Pos1, Pos2).
otherwise.
init_pos(-1, _, _, _, Con1, Con3, Pos, Pos2) :- true | Con1 = Con3, Pos = Pos2.
```

Eventually, the following predicate consolidates the information in the connection and cost matrices: the cost of an edge is put after the label of the referred vertex.

```
consolidate(Con1, Cost, Con2) :-
    vector(Con1, _), vector(Cost, Size) |
        consolidate(~(Size-1), Con1, Cost, Con2).

consolidate(P, Con1, Cost, Con4) :-
    P > -1 |
        set_vector_element(Con1, P, Old, _, Con2),
        set_vector_element(Cost, P, OldC, _, Cost1),
        consolidate_string(Old, OldC, New),
        (wait(New) ->
            set_vector_element(Con2, P, _, New, Con3),
            consolidate(~(P-1), Con3, Cost1, Con4)).
otherwise.
consolidate(_, Con1, _, Con2) :- true | Con2 = Con1.

consolidate_string(Old, OldC, New) :-
    string(Old, Size, _) |
        new_string(New0, ~(2*Size), 32),
        consolidate_string(~(Size-1), Old, OldC, New0, New).

consolidate_string(P, Old, OldC, New0, New3) :-
    P > -1 |
```

```
        string_element(Old, P, E1, Old1),
        string_element(OldC, P, E2, OldC1),
        set_string_element(New0, ~(2*P), E1, New1),
        set_string_element(New1, ~(2*P+1), E2, New2),
        consolidate_string(~(P-1), Old1, OldC1, New2, New3).
otherwise.
consolidate_string(_, _, _, New0, New3) :- true | New0 = New3.
```

2.3 Start of parallel work and control

Here, we start the parallel processing. Each processor receives a copy of the connection matrix, the number of sets and the number of processors. Then, a control predicate is called, which iterates the Lin-Kernighan algorithm until a minimum of cost is reached.

```
start_op(Con, Set, Pos0, UsedProc, Pos) :-
    vector(Con, _), integer(Set), string(Pos0, _, _), UsedProc >= 1 |
        util:object_to_string({Con, Set, UsedProc}, _, Data),
        par:sync_create_2_tree_of_p(0, ~(UsedProc-1), Data, Tree),
        par:apply_2_tree_of_p(Tree, _, tree, node,
                              {data_in, rank_in, children_in,
                               tree_up_in, tree_up_out,
                               tree_left_in, tree_left_out,
                               tree_right_in, tree_right_out,
                               processor_in},
                              _, _, [], T0, [Pos0 | TI], _, []),
        control(T0, TI, Pos).
```

The control predicates sends the position string to the tree of processes. This starts the Lin-Kernighan algorithm. Whenever the algorithm finishes, a position string is received, as well as a flag. If flag value is 1, some change occured, which improved the partition. In this case, the algorithm is restarted. Otherwise, flag equals 0: optimization can be stopped and the current position be returned through the arguement **Pos**.

```
control([0, Pos0 | _], TI, Pos) :-
    string(Pos0, _, _) |
    Pos = Pos0, TI = [].
otherwise.
control([_, Pos0 | T0], TI, Pos) :-
    string(Pos0, _, _) |
        TI = [Pos0 | TI1],
        control(T0, TI1, Pos).
```

3 Module tree

Distribution of work load is done over several processors, organized as a binary tree, with respect to communications.

This module contains the predicates defining the basic distribution of the algorithm, that is, the synchronization with the top driving process and the collation of permutations together until the head of the tree. It also holds the algorithm which determines which pair is treated by a processor.

```
:- module tree.
:- public node/10.
```

The core of the communication algorithm is quite simple. In any given pass, the position string is propagated down the tree, computation takes place, and a string containing permutation is built and propagated from the bottom of the tree. A new position string is derived from these permutations and the process iterates.

To determine which pair is associated to one processor is tricky. In the first place, let's assume that the number of processor is not bounded. If we have s sets, there cannot be, at any given time, more than $s/2$ pairs treated in parallel. Our problem is therefore to generate s sequences of $s/2$ pairs, as the total number of pairs we can take among s sets is $\frac{s(s-1)}{2}$. Two cases are distinguished:

- If $s = 2k + 1, k \in \mathcal{N}$, and if we assume sets are labelled as $0, \ldots, 2k$, we consider the pairs $(i,j)_{m,n}$, $m \in [0, k-1]$, and $n \in [0, s-1]$, such that $(i,j)_{m,n} \equiv ((m+n)(s), (s-m+n-2)(s))$. Then, for a given n_0, the best permutations between 2 sets, in the sense of the Lin-Kernighan algorithm, can be computed in parallel for the k set-pairs $(i,j)_{m,n_0}$, for $m \in [0, k-1]$. Let's call such a set of pair I_{n_0}.
  All I_n should be enumerated an computed. Since set labels are not distincts between two I_n's, the best permutations for 2 I_n's cannot be performed concurrently.
  The maximum degree of parallelism thus achieved is k.

- For an even $s = 2k, k \in \mathcal{N}$, we have to consider 2 different types of pairs:

  1. $(i,j)_{m,n} \equiv ((m+n)(s), (s-m+n-2)(s))$, for $m \in [0, k-2]$, $n \in [0, k-1]$;
  2. $(i',j')_{m,n} \equiv ((m+n)(s), (s-m+n-1)(s))$, for $m \in [0, k-1]$, $n \in [0, k-1]$.

  Like in the previous case, permutations can be computed in parallel for the pairs I_n: $(i,j)_{m,n}$ for $m \in [0, k-2]$, and for the pairs I'_n: $(i',j')_{m,n}$ for $m \in [0, k-1]$. But computations for different I'_n's and I_n's cannot be performed concurrently.
  The maximum degree of parallelism achieved in this case is $k - 0.5$.

If the number of available processors is $p < k$, we chose to decompose I_n or I'_n into subsets of p pairs. Of course, this is not optimal if p is not a divider of k. The achieved parallelism is in the worst case $p(1 - p/k)$.

3.1 For a node in the tree...

The following predicate waits for the data describing the graph to be ready. Then, in the case of the head of the tree, inserts a process (predicate `head/5`) in the communicating stream, which will transfer the position string coming from the top driving process, on one hand. On the other hand, this process with take the permutation string coming from the bottom of the tree and alter the position string accordingly.

Depending on whether or not the current processor is the head of the tree, the following predicate starts a communication process, which waits for a position string, starts the search for the best permutations, then outputs a string with those permutations.

```
node(CData, Rank, Child, TUI, TUO, TLI, TLO, TRI, TRO, Proc) :-
   string(CData,_,_), integer(Rank), integer(Child), integer(Proc) |
        util:string_to_object(CData, _, Data),
```

```
        (wait(Data), Rank = 0 ->
            Data = {Con, Set, UsedProc},
            head(Set, TUI, TU0, TUI1, TU01),
            com(Child, {0, Proc, Con, Set, UsedProc}, % 0 = pass number
                {TUI1, TU01, TLI, TL0, TRI, TR0});
         wait(Data), Rank > 0 ->
            Data = {Con, Set, UsedProc},
            com(Child, {0, Proc, Con, Set, UsedProc}, % 0 = pass number
                {TUI, TU0, TLI, TL0, TRI, TR0})).
```

3.2 Head of the tree

The following process propagates the position information and collates permutations together. It also keeps up the number of passes which are necessary to complete the algorithm. During each pass, a different pair of sets is selected by each processor. If there are s sets, there are s passes, during which $s/2$ or $s/2-1$ pairs are processed, depending on the parity of s.

The following predicate waits for a request from the process `control/3` in module `lk`. Note also that the number of passes is different from s when $s = 2$. In this case, only one pass is necessary

```
head(Set, [Pos | TUI], TU0, TUI1, TU01) :-
    Set >= 2, string(Pos, _, _) |
        util:sync_copy(Pos, Pos1, Pos2),
        (wait(Pos2) ->
            TUI1 = [Pos2 | TUI2],
            wait_perms(TUI, TU0, TUI2, TU01, Pos1, Set, Set, 0)).
otherwise.
head(_, [], TU0, TUI1, _) :- true | TU0 = [], TUI1 = [].
```

The following predicate accepts message from the tree of processors, and updates accordingly the position string and the modification flag.

```
wait_perms(TUI, TU0, TUI1, [Perms | TU01], Pos, Set, Pass, Modif) :-
    string(Perms, 0, _) |
        check_pass(TUI, TU0, TUI1, TU01, Pos, Set, Pass, Modif).
otherwise.
wait_perms(TUI, TU0, TUI1, [Perms | TU01], Pos, Set, Pass, _) :-
    string(Perms, Size, _) |
        apply_perms(~(Size-1), Perms, Pos, Pos1),
        (wait(Pos1) ->
            check_pass(TUI, TU0, TUI1, TU01, Pos1, Set, Pass, 1)).
```

In the following predicate, in the case of the last pass, we indicate to the top predicate (`control/3` in module `lk`) whether a change occured or not, and we wait for a new request, through the predicate `head/5`. Otherwise, we iterate through the next pass, giving a copy of the current position string to the process tree.

```
check_pass(TUI, TU0, TUI1, TU01, Pos, Set, Pass, Modif) :-
    Pass > 1 |
        util:sync_copy(Pos, Pos1, Pos2),
        (wait(Pos2) ->
            TUI1 = [Pos1 | TUI2],
            wait_perms(TUI, TU0, TUI2, TU01, Pos2, Set, ~(Pass-1), Modif)).
otherwise.
```

```
check_pass(TUI, TU0, TUI1, TU01, Pos, Set, _, Modif) :- true |
        TU0 = [Modif, Pos | TU02],
        head(Set, TUI, TU02, TUI1, TU01).
```

Last of this section devoted to the tree head, the following predicate applies upon the position string the permutations computed by application of the Lin Kernighan algorithm.

```
apply_perms(P, Perms, Pos, Pos3) :-
    P > 0,
    string_element(Perms, P, E1), string_element(Perms, ^(P-1), E2),
    string_element(Pos, E1, P1), string_element(Pos, E2, P2) |
        set_string_element(Pos, E1, P2, Pos1),
        set_string_element(Pos1, E2, P1, Pos2),
        apply_perms(^(P-2), Perms, Pos2, Pos3).
otherwise.
apply_perms(_, _, Pos, Pos3) :- true | Pos = Pos3.
```

3.3 Communication process

The following predicate, according to the number of children of the current node, selects the proper communication process.

```
com(2, Data, {TUI, TU0, TLI, TL0, TRI, TR0}) :- true |
        full_com(TUI, TU0, TLI, TL0, TRI, TR0, Data).
com(1, Data, {TUI, TU0, TLI, TL0, TRI, TR0}) :- true |
        left_com(TUI, TU0, TLI, TL0, TRI, TR0, Data).
com(0, Data, {TUI, TU0, TLI, TL0, TRI, TR0}) :- true |
        one_com(TUI, TU0, TLI, TL0, TRI, TR0, Data).
```

The following predicate implements the communication process in the case of a node with 2 children. Communication is decomposed into 2 distincts phases:

- first, we duplicate the position information coming from the top: this is done in the predicate **full_com/7**.
- then, we wait for the permutations coming from the local process and the subtrees, and append these strings, which are sent to the parent node. This is done in the predicate **full_com2/8**.

```
full_com([Pos | TUI], TU0, TLI, TL0, TRI, TR0, Data) :-
    string(Pos, _, _) |
        util:sync_copy(Pos, Pos1, Pos2),
        util:sync_copy(Pos1, Pos3, Pos4),
        (string(Pos4, _, _) ->
            TL0 = [Pos2 | TL01], TR0 = [Pos3 | TR01],
            one_pair(Pos4, Data, Data1, Per)@priority($,-2000),
            full_com2(TUI, TU0, TLI, TL01, TRI, TR01, Data1, Per)).
full_com([], _, _, TL0, _, TR0, _) :- true | TL0 = [], TR0 = [].

full_com2(TUI, TU0, [Per1 | TLI], TL0, [Per2 | TRI], TR0, Data1, Per) :-
    string(Per1, _, _), string(Per2, _, _), string(Per, _, _) |
        fel:appendstring([Per, Per1, Per2], Per0),
        (wait(Per0) ->
            TU0 = [Per0 | TU01],
            full_com(TUI, TU01, TLI, TL0, TRI, TR0, Data1)).
```

The following predicates do as above for the case when there is only the left children.

```
left_com([Pos | TUI], TU0, TLI, TL0, _, _, Data) :-
    string(Pos, _, _) |
        util:sync_copy(Pos, Pos1, Pos2),
        (string(Pos2, _, _) ->
            TL0 = [Pos2 | TL01],
            one_pair(Pos1, Data, Data1, Per)@priority($,-2000),
            left_com2(TUI, TU0, TLI, TL01, _, _, Data1, Per)).
left_com([], _, _, TL0, _, _, _) :- true | TL0 = [].

left_com2(TUI, TU0, [Per1 | TLI], TL0, _, _, Data1, Per) :-
    string(Per1, _, _), string(Per, _, _) |
        fel:appendstring(Per, Per1, Per0),
        (wait(Per0) ->
            TU0 = [Per0 | TU01],
            left_com(TUI, TU01, TLI, TL0, _, _, Data1)).
```

Eventually, we do in the following predicate with the case of a leaf node.

```
one_com([Pos | TUI], TU0, _, _, _, _, Data) :-
    string(Pos, _, _) |
        one_pair(Pos, Data, Data1, Per),
        one_com2(TUI, TU0, _, _, _, _, Data1, Per).
one_com([], _, _, _, _, _, _) :- true | true.

one_com2(TUI, TU0, _, _, _, _, Data1, Per) :-
    string(Per, _, _) |
        TU0 = [Per | TU01],
        one_com(TUI, TU01, _, _, _, _, Data1).
```

3.4 Pair determination

In this section, we determine which pair to use and how many pair to use (if there are not enough processors). This is done according to the current processor number, to the pass number and to the parity.

The following predicate starts the creation of a list of set pairs; several permutations can be computed on the same processor within the same pass if `UsedProc` is less than half of the number of sets. Then, the Lin-Kernighan algorithm is called.

```
one_pair(Pos, {Pass, Proc, Con, Set, UsedProc}, Data1, Per) :-
    integer(Pass), integer(Proc), integer(Set), integer(UsedProc) |
        make_pair(Pass, Proc, Set, UsedProc, List, List, Pairs),
        algo:lk(Pairs, Set, Pos, Con, Con1, Per1),
        (string(Per1, _, _) ->
            Data1 = {~(Pass+1), Proc, Con1, Set, UsedProc},
            Per = Per1).
```

The following predicates contruct the list of pairs. Two cases are distinguished first:

- If the number of sets is odd, there is a single way to construct pairs, and $s-1$ circular permutations are done over the set list.

- If the number of sets is even, there are two types of pairs: the ones constructed as above and the one constructed with an odd distance between sets in a pair. There are $s^2/4$ pairs of the later type, which are constructed first, and $s(s-2)/4$ pairs of the former type, which are constructed last in the following predicate.

```
make_pair(Pass, Proc, Set, UsedProc, List, Head, Pairs) :-
    Set mod 2 =:= 1 |
        make_pair_odd(Pass, Proc, Set, UsedProc, List, Head, Pairs).
otherwise.
make_pair(Pass, Proc, Set, UsedProc, List, Head, Pairs) :- true |
        make_pair_even(Pass, Proc, Set, UsedProc, List, Head, Pairs).
```

Here is the predicate for an odd s.

```
make_pair_odd(Pass, Proc, Set, UsedProc, List, Head, Pairs) :-
   Proc < Set / 2 |
        I := (Pass+Proc) mod Set,
        J := (Set-2+Pass-Proc) mod Set,
        fel:mini(I,J,Min),
        fel:maxi(I,J,Max),
        (wait(Min), wait(Max) ->
            List = [Min, Max | List1],
            make_pair_odd(Pass, ~(Proc+UsedProc), Set, UsedProc,
                      List1, Head, Pairs)).
otherwise.
make_pair_odd(_, _, _, _, List, Head, Pairs) :- true |
        List = [], Pairs = Head.
```

Here are the two predicates for an even s.

```
make_pair_even(Pass, Proc, Set, UsedProc, List, Head, Pairs) :-
    Pass < Set / 2, Proc < Set / 2 |
        I := (Pass+Proc) mod Set,
        J := (Set-1+Pass-Proc) mod Set,
        fel:mini(I,J,Min),
        fel:maxi(I,J,Max),
        (wait(Min), wait(Max) ->
            List = [Min, Max | List1],
            make_pair_even(Pass, ~(Proc+UsedProc), Set, UsedProc,
                      List1, Head, Pairs)).
otherwise.
make_pair_even(Pass, _, Set, _, List, Head, Pairs) :-
    Pass < Set / 2 |
        List = [], Pairs = Head.
otherwise.
make_pair_even(Pass, Proc, Set, UsedProc, List, Head, Pairs) :- true |
        make_pair_even2(Pass, Proc, Set, UsedProc, List, Head, Pairs).

make_pair_even2(Pass, Proc, Set, UsedProc, List, Head, Pairs) :-
    Proc < (Set / 2) - 1 |
        I := (Pass-(Set/2)+Proc) mod Set,
        J := (Set-2+Pass-(Set/2)-Proc) mod Set,
```

```
        fel:mini(I,J,Min),
        fel:maxi(I,J,Max),
        (wait(Min), wait(Max) ->
            List = [Min, Max | List1],
            make_pair_even2(Pass, ~(Proc+UsedProc), Set, UsedProc,
                      List1, Head, Pairs)).
otherwise.
make_pair_even2(_, _, _, _, List, Head, Pairs) :- true |
        List = [], Pairs = Head.
```

4 Module algo

This module holds the core of the Lin-Kernighan algorithm, to find the best permutation of elements between a specified list of set pairs. The basic algorithm is detailed in [2], and we follow it closely. Some particular points of this implementation are detailled here after.

```
:- module algo.
:- public lk/6.
```

4.1 From a list of set-pairs...

The following (top-level) predicate is called from the module `tree` in order to find the best permutations, for a list of set pairs contained in `Pairs`. Our careful reader has already noticed that even on the same processor, several applications of the Lin-Kernighan algorithm were done with the same set of data. The only varying parameter is the list of pairs. That means that no intermediary structure is kept from one computation to the next one. Connection-cost matrix remains of course untouched.

In the following predicate, we build the intermediary structure, then we invoke the Lin-Kernighan resolution over the list of set pairs, to get back a list of vertex permutations, with respect to set inclusion, which is transformed into a string. This string is used by the calling predicate.

```
lk(Pairs, Set, Pos, Con, Con1, Per1) :-
    list(Pairs), integer(Set), string(Pos, Size, _),
    vector(Con, _) |
        make_intermediary(Pairs, Pairs1, Set, Pos, Con, Con1, Inter),
        pair_lk(Pairs1, ~(Size/Set), Inter, Per),
        list:list_to_string(Per, 32, Per1).
lk([], _, _, Con, Con1, Per1) :-
    vector(Con, _) |
    new_string(Per1, 0, 32),
    Con = Con1.
```

4.2 Building the intermediary structure

An intermediary structure has to be built to satisfy speed requirements of the Lin-Kernighan algorithm. The connection and cost matrices are convenient for transfer because they are small, but they are not practical when we want to access directly the cost value of the connection between two arbitrary nodes.

For each set concerned in the optimization, we need first of all the list of vertices in the set. Then, for each vertex, we need the list of connected vertices, in the current set and in the other set (of the relevant pair). Therefore, we build, for each set in the pair list, a string containing $2n^2 + n$ elements, where n is the number of elements per set. The label of the ith vertex is stored at position i in the string. We now consider connections between this set S_1 and some other given set S_2. That set also has n elements, which can be indexed similarly by j, between 1 and n. Cost of the edge from i to i', both in S_1, can be found at position $(2 \times i - 1)n + i'$ in the string. Cost of edge from i to j, the later in S_2, can be found at position $2i \times n + j$.

Since those strings are expected to change from one pass to the other, we don't keep them, nor try to share them between processors. We think (it's not proven) that keeping them up incrementally or exchanging them between processors would be slower than systematic re-generation.

The following predicate scans the list of set pairs and creates an intermediary structure for each set in the list. In the first place, we scan the pair list to check which sets are concerned and we initialize the intermediary structure. Also, we create an index from the set of vertex labels to their relative position in the set-to-vertex string. Then, we scan the vertex to set affectation string to fill up the n first characters of the intermediary structure strings. Eventually, we insert the cost-connection information.

```
make_intermediary(Pairs1, Pairs3, Set, Pos, Con, Con1, Inter3) :-
    string(Pos, Size, _), vector(Con, _) |
        new_vector(Inter, Set),
        new_string(Index, Size, 32),
        SetSize := Size / Set,
        StringSize := SetSize*(1 + 2 * SetSize),
        scan_list(Pairs1, Pairs2, ~(SetSize-1), StringSize, Inter, Inter1),
        fill_inter(~(Size-1), Pos, Pos1, Inter1, Inter2,
                   Index, Index1),
        cost_inter(Pairs2, Pairs3, SetSize, Pos1,
                   Index1, Inter2, Inter3, Con, Con1).
```

The following predicate scans the pair list and initializes the intermediary structure with strings of the proper length, at the position corresponding to sets in the list.

```
scan_list([S1, S2 | Pairs], Pairs1, Size, SS, Inter, Inter3) :-
    integer(S1), integer(S2) |
        new_string(Con1, SS, 32),
        set_string_element(Con1, 0, Size, Con2),
        set_vector_element(Inter, S1, _, Con2, Inter1),
        new_string(Con3, SS, 32),
        set_string_element(Con3, 0, Size, Con4),
        set_vector_element(Inter1, S2, _, Con4, Inter2),
        Pairs1 = [S1, S2 | Pairs2],
        scan_list(Pairs, Pairs2, Size, SS, Inter2, Inter3).
otherwise.
scan_list(Pairs, Pairs1, _, _, Inter, Inter2) :- true |
        Pairs = Pairs1, Inter = Inter2.
```

The following predicate scans the **Pos** string and inserts in the relevant string, from the intermediary structure, the label of each vertex which is found to be included in some set of interest. In each string, the character at position 0 is used to store the current number of vertices actually stored in the string. We also set in the string **index** the position at which each vertex is put.

```
fill_inter(P, Pos, Pos1, Inter1, Inter2, Index, Index2) :-
    P > -1, string_element(Pos, P, E), vector_element(Inter1, E, 0) |
        fill_inter(~(P-1), Pos, Pos1, Inter1, Inter2, Index, Index2).
otherwise.
fill_inter(P, Pos, Pos1, Inter1, Inter3, Index, Index2) :-
    P > -1, string_element(Pos, P, E) |
        set_vector_element(Inter1, E, S1, S4, Inter2),
        string_element(S1, 0, N, S2),
        set_string_element(S2, 0, ~(N-1), S3),
        set_string_element(S3, N, P, S4),
        set_string_element(Index, P, N, Index1),
        fill_inter(~(P-1), Pos, Pos1, Inter2, Inter3,
                   Index1, Index2).
otherwise.
fill_inter(_, Pos, Pos1, Inter1, Inter2, Index, Index2) :- true |
    Pos = Pos1, Inter1 = Inter2, Index = Index2.
```

The following predicate updates the intermediary structure for a list of pair of sets, inserting cost information.

```
cost_inter([S1, S2 | Pairs], Pairs1, SetSize, Pos,
           Index, Inter1, Inter4, Con1, Con4) :-
    integer(S1), integer(S2), integer(SetSize), string(Index, _, _),
    vector(Inter1, _), vector(Con1, _), string(Pos, _, _) |
        set_vector_element(Inter1, S1, OldS1, NewS1, Inter2),
        cost_inter2(~(SetSize-1), SetSize, S1, S2,
                    Pos, Pos1, Index, Index1, OldS1, NewS1, Con1, Con2),
        set_vector_element(Inter2, S2, OldS2, NewS2, Inter3),
        cost_inter2(~(SetSize-1), SetSize, S2, S1,
                    Pos1, Pos2, Index1, Index2, OldS2, NewS2, Con2, Con3),
        Pairs1 = [S1, S2 | Pairs2],
        cost_inter(Pairs, Pairs2, SetSize, Pos2, Index2, Inter3, Inter4,
           Con3, Con4).
cost_inter([], Pairs1, _, _, _, Inter1, Inter4, Con1, Con4) :- true |
        Pairs1 = [], Inter1 = Inter4, Con1 = Con4.
```

The following predicate inserts the cost information in the part of the intermediary structure related to the set S1. The basic idea is to scan all the vertices in S1 and to pick up the cost information whenever its related to a node either in S1 or in S2.

```
cost_inter2(P, SetSize, S1, S2, Pos, Pos2, Index, Index2, String1, String3,
            Con1, Con3) :-
    string_element(String1, P, Vert) |
        set_vector_element(Con1, Vert, OldCon, NewCon, Con2),
        update_inter(SetSize, P, OldCon, NewCon, S1, S2, String1, String2,
                     Pos, Pos1, Index, Index1),
        cost_inter2(~(P-1), SetSize, S1, S2, Pos1, Pos2, Index1, Index2,
                    String2, String3, Con2, Con3).
otherwise.
cost_inter2(_, _, _, _, Pos, Pos1, Index, Index2, String1, String3,
           Con1, Con3) :-
    string(String1, _, _) |
        Pos = Pos1, Index = Index2, String1 = String3, Con1 = Con3.
```

The following predicate scans the list of vertices, and if they belong to S1 or S2, they are put in corresponding places in the intermediary structure string, using the index string to find the relative position of a vertex in the set.

```
update_inter(SetSize, I, OldCon, NewCon, S1, S2, Str1, Str2, Pos, Pos1,
             Ind, Ind1) :-
    string(OldCon, Size, _) |
        update_inter(I, ~(Size-2), SetSize, S1, S2, OldCon, NewCon, Str1, Str2,
                     Pos, Pos1, Ind, Ind1).

update_inter(I, P, SetSize, S1, S2, OldCon, NewCon, Str1, Str3,
             Pos, Pos1, Ind, Ind2) :-
    string_element(OldCon, P, El), string_element(Pos, El, S1) |
        string_element(Ind, El, J, Ind1),
        string_element(OldCon, ~(P+1), Cost, OldCon1),
        set_string_element(Str1, ~(SetSize*(1+2*I)+J), Cost, Str2),
        update_inter(I, ~(P-2), SetSize, S1, S2, OldCon1, NewCon, Str2, Str3,
                     Pos, Pos1, Ind1, Ind2).
```

```
update_inter(I, P, SetSize, S1, S2, OldCon, NewCon, Str1, Str3,
             Pos, Pos1, Ind, Ind2) :-
    string_element(OldCon, P, El), string_element(Pos, El, S2) |
        string_element(Ind, El, J, Ind1),
        string_element(OldCon, ~(P+1), Cost, OldCon1),
        set_string_element(Str1, ~(2*SetSize*(I+1)+J), Cost, Str2),
        update_inter(I, ~(P-2), SetSize, S1, S2, OldCon1, NewCon, Str2, Str3,
                     Pos, Pos1, Ind1, Ind2).
otherwise.
update_inter(I, P, SetSize, S1, S2, OldCon, NewCon, Str1, Str3,
             Pos, Pos1, Ind, Ind2) :-
    P > -1 |
        update_inter(I, ~(P-2), SetSize, S1, S2, OldCon, NewCon, Str1, Str3,
                     Pos, Pos1, Ind, Ind2).
otherwise.
update_inter(_, _, _, _, _, OldCon, NewCon, Str1, Str3, Pos, Pos1,
             Ind, Ind2) :- true |
        OldCon = NewCon, Str1 = Str3, Pos = Pos1, Ind = Ind2.
```

4.3 Lin-Kernighan algorithm

Here is the biggy. Let's recall quickly how this algorithm works. At any time, we look for the best permutation between two vertices in the 2 sets of interest. We permute and freeze these nodes. We iterate the process until only frozen nodes remain. Now, all nodes have been exchanged. If we look at the cumulated score of permutations, considering connection cost, the score reaches a maximum. At the end, the score is of indeed zero, since sets contents have been exchanged. We keep the permutations leading to the maximum score.

The following predicate applies the Lin-Algorithm to a list of pairs of sets.

```
pair_lk([S1, S2 | Pairs], Size, Inter, Per) :-
    integer(S1), integer(S2), integer(Size), vector(Inter, _) |
        one_lk(S1, S2, Size, Inter, Inter1, Per, Per1),
        pair_lk(Pairs, Size, Inter1, Per1).
otherwise.
pair_lk(_, _, _, Per) :- true | Per = [].
```

Now, we have to consider the problem of finding the best permutations for a simple pair of sets. At the top level, in the first place we build the gain lists associated to the two sets. Then, we do all possible permutations and select the segment of the permutation sequence which brings the largest gain. When no permutation brings any improvement, this segment is empty, by definition.

```
one_lk(S1, S2, Size, Inter, Inter4, StartPer, EndPer) :-
    integer(S1), integer(S2), vector(Inter, _) |
        set_vector_element(Inter, S1, Cost1, _, Inter1),
        set_vector_element(Inter1, S2, Cost2, _, Inter2),
        build_list(~(Size-1), Size, Cost1, Cost11, D1),
        build_list(~(Size-1), Size, Cost2, Cost21, D2),
        perms_lk(Size, D1, D2, Cost11, Cost12, Cost21, Cost22, Size, CostPer),
        cut_at_best(CostPer, ListPer, 0, 0, 0, 0, Cut),
        list:split(ListPer, Cut, StartPer, EndPer, Sync),
        (wait(Cost12), wait(Cost22), wait(Sync) ->
            set_vector_element(Inter2, S1, _, Cost12, Inter3),
```

```
        set_vector_element(Inter3, S2, _, Cost22, Inter4)).
```

The following predicate builds a list of possible gain in cutting cost. Each gain D_i is associated to a node in a set. This gain is the one which would occur if the node was displaced alone from the current set to the other set of the pair.

```
build_list(P, Size, Cost1, Cost4, D1) :-
    P > -1, integer(Size), string(Cost1, _, _) |
        Start := Size + P*2*Size,
        End := Start + Size,
        sum_cost(Start, End, Cost1, Cost2, 0, E1), % local set cost
        sum_cost(End, ~(End+Size), Cost2, Cost3, 0, E2), % opposite set cost
        D1 = [~(E2-E1), P | D2],
        build_list(~(P-1), Size, Cost3, Cost4, D2).
otherwise.
build_list(_, _, Cost1, Cost4, D1) :- true | Cost1 = Cost4, D1 = [].
```

The following predicate sums the cost from position P1 until position P2.

```
sum_cost(Start, End, Cost1, Cost2, E, E1) :-
    Start < End, string_element(Cost1, Start, C) |
        sum_cost(~(Start+1), End, Cost1, Cost2, ~(E+C), E1).
otherwise.
sum_cost(_, _, Cost1, Cost2, E, E1) :- true | Cost1 = Cost2, E = E1.
```

The cost-permutation list returned by **perms_1k** is a list of 3-uples, each one holding the gain associated to the permutation and the labels of the vertices to permute. The following predicate takes this list and returns the list of the permutations without the cost information, and the number of elements to keep in the list to get the optimal set of permutations.

```
cut_at_best([Cost, P1, P2 | CPer], LPer, Score, Best, CCut, _, Res) :-
    NScore := Score + Cost,
    NScore > Best |
        LPer = [P1, P2 | LPer2],
        NCut := CCut + 2,
        cut_at_best(CPer, LPer2, NScore, NScore, NCut, NCut, Res).
otherwise.
cut_at_best([Cost, P1, P2 | CPer], LPer, Score, Best, CCut, Cut, Res) :- true |
        NScore := Score + Cost,
        LPer = [P1, P2 | LPer2],
        NCut := CCut + 2,
        cut_at_best(CPer, LPer2, NScore, Best, NCut, Cut, Res).
otherwise.
cut_at_best(CPer, LPer, _, _, _, Cut, Res) :- true |
        CPer = LPer, Cut = Res.
```

4.4 Searching the gain lists

A naive algorithm would try all permutations and keep the best. That would take $\mathcal{O}(n^2)$ operations, and since this has to be done until n permutations are found, that would require all in all $\mathcal{O}(n^3)$ permutations. There is a basic optimization if we sort the gain lists D1 and D2. Suppose we have gains A_i in the first list

and B_i in the second. We can go through all pairs (A_i, B_j) in decreasing order of $A_i + B_j$. The real gain achieved if we permute i and j is $A_i + B_j - 2c_{ij}$, where c_{ij} is the cost of the edge between i and j. Since all costs are positive, we can stop the search for the optimal permutation as soon as we encounter a pair (i, j) such that $A_i + B_j$ is less than the real gain achieved so far. In this case, the dominant complexity factor is the one associated to sort. All in all, complexity becomes $\mathcal{O}(n^2 \log n)$.

To effectively implement this algorithm, we have first to sort each list, then to go through permutations in an ordered manner. This is done by the following predicate, which works until the gain lists are empty. That way, vertices which have been subject to a permutation are implicitely taken away from the search space, for further computations. At the end of the following predicate, before we go with the remaining of the list, note that we purge the gain lists from the permuted vertices and update gain values to reflect the permutation.

```
perms_lk(LSize, D1, D2, Cost11, Cost14, Cost21, Cost24, Size, CostPer) :-
    LSize > 0, list(D1), list(D2),
    string(Cost11, _, _), string(Cost21, _, _), integer(Size) |
        gain_sort(LSize, D1, D11),
        gain_sort(LSize, D2, D21),
        list:list_to_string(D11, 32, S11),
        list:list_to_string(D21, 32, S21),
        search_perms(Size, S11, S12, S21, S22, Cost11, Cost12, Cost21, Cost22, Per),
        gain_filter(CostPer, Per, CostPer2, S12, D12, S22, D22,
                    Cost12, Cost13, Cost22, Cost23),
        perms_lk(~(LSize-1), D12, D22, Cost13, Cost14, Cost23, Cost24,
                 Size, CostPer2).
otherwise.
perms_lk(0, [], [], Cost11, Cost12, Cost21, Cost22, _, CostPer) :-
    string(Cost11, _, _), string(Cost21, _, _) |
        Cost11 = Cost12, Cost21 = Cost22, CostPer = [].
```

The two following predicates perform a merge-sort for lists. The only reason why we did not use FLIB is that the sort key and the element position in the set are at the same level. We could have arranged that by encapsulating them together in a vector or a list, but sort speed would be quite slower. Be aware that in the following, LSize is half of the number of elements in the list, i.e. it holds the number of pairs to sort.

```
gain_sort(LSize, D11, D13) :-
    LSize > 2, list(D11) |
        Size1 := LSize >> 1,
        list:sync_split(D11, ~(Size1 << 1), D21, [], D31),
        gain_sort(Size1, D21, D22),
        gain_sort(~(LSize - Size1), D31, D32),
        gain_merge(D22, D32, D12, Sync),
        (wait(Sync) -> D12 = D13).
otherwise.
gain_sort(2, [C1, P1, C2, P2], D) :- C1 < C2 | D = [C2, P2, C1, P1].
otherwise.
gain_sort(_, D1, D2) :- true | D1 = D2.

gain_merge([C1, P1 | D1], [C2, P2 | D2], D3, Sync) :-
    C1 >= C2 |
        D3 = [C1, P1 | D4],
        gain_merge(D1, [C2, P2 | D2], D4, Sync).
otherwise.
gain_merge([C1, P1 | D1], [C2, P2 | D2], D3, Sync) :-
```

```
    C2 >= C1 |
        D3 = [C2, P2 | D4],
        gain_merge([C1, P1 | D1], D2, D4, Sync).
otherwise.
gain_merge([], D2, D3, Sync) :- true | D3 = D2, Sync = 1.
otherwise.
gain_merge(D1, [], D3, Sync) :- true | D3 = D1, Sync = 1.
```

Now, let's deal with the filtering of permuted nodes. We also transform the relative position of vertices in sets into global labels.

```
gain_filter(CostPer, [C, P1, P2], CostPer2, S11, D11, S21, D21,
            Cost12, Cost13, Cost22, Cost23) :-
    integer(P1), integer(P2),
    string(S11, Size, _), string(S21, Size, _) |
        string_filter(^(Size-1), P1, S11, D1, Sync1),
        string_filter(^(Size-1), P2, S21, D2, Sync2),
        string_element(Cost12, P1, L1, Cost13),
        string_element(Cost22, P2, L2, Cost23),
        (wait(Sync2), wait(Sync1) ->
            D1 = D11, D2 = D21,
            CostPer = [C, L1, L2 | CostPer2]).

string_filter(P, P1, S1, D1, Sync1) :-
    P > -1, string_element(S1, P, P1) |
        string_filter(^(P-2), P1, S1, D1, Sync1).
otherwise.
string_filter(P, P1, S1, D1, Sync1) :-
    P > -1, string_element(S1, P, E1), string_element(S1, ^(P-1), E2) |
        D1 = [E2, E1 | D2],
        string_filter(^(P-2), P1, S1, D2, Sync1).
otherwise.
string_filter(_, _, _, D1, Sync1) :- true | D1 = [], Sync1 = 1.
```

4.5 The Quest for the best permutation

We look for the best permutation, scanning the gain list. We have to scan then so that the sum of gains in the elected pair monotonously decreases.

In the following predicate, we compute an initial gain, then call the list scan predicate. After a permutation has been done, we have to correct the gain value in the remaining part of the list.

```
search_perms(Size, S11, S13, S21, S24, Cost11, Cost13, Cost21, Cost23, Per) :-
    string(S11, El1, _), string(S21, El2, _),
    string(Cost11, _, _), string(Cost21, _, _),
    integer(Size) |
        string_element(S11, 1, El1, S12),
        string_element(S21, 0, G, S22),
        string_element(S22, 1, El2, S23),
        string_element(Cost11, ^(2*Size*(El1+1)+El2), Cfs, Cost12),
        G1 := G - Cfs,
        string_element(Cost21, ^(2*Size*(El2+1)+El1), Csf, Cost22),
        G2 := G1 - Csf,
```

```
search_perms(2, Els, El1, G2, S23,
             {S12, Size, Cost12, Cost22, 0, 0, 0}, Final),
wait_perms(Final, Els, S13, S24, Cost13, Cost23, Per).
```

We now scan the pair lists. `First` is the position of the pivot in the first list. All other elements in the other list are tried. If no potential gain is found when taking care of the first element of this list, the search is over. Otherwise, the search in the second list is only stopped and the pivot is incremented. This is it for the principle. Implementation is a little bit painful, because many variables need to be at hand. Therefore, vector `Costs` holds many things. Namely:

```
Costs = {S1, Size, Cost1, Cost2, P1, P2, First}
```

`S1` is the string holding the gain list for the first set. `Size` is the size of each set. This is useful to access the costs information in `Cost1` and `Cost2`. `P1` and `P2` are the position of the pivot and second element of the pair having the best gain so far. `First` is the position of the current pivot.

In the following predicate, to gain time, we do not compare full gains but only the variation of gain due to the second element. This variation is stored in the variable `SecGain`. On the other hand, we have to correct this "best" variation whenever we change the pivot and at the end of the computation.

The following predicate works according to 3 different patterns:

- In the first case, for all elements in the second list between position `Second` and position `EndSecond` we check that there is a potential gain over `SecGain`.

```
search_perms(Second, EndSecond, El1, SecGain, S21, Costs1, Costs5) :-
     Second < EndSecond, string_element(S21, Second, G), G > SecGain |
        string_element(S21, ~(Second+1), El2, S22),
        set_vector_element(Costs1, 2, Cost11, Cost12, Costs2),
        vector_element(Costs2, 1, Size, Costs3),
        string_element(Cost11, ~(2*Size*(El1+1)+El2), Cfs, Cost12),
        V := G - Cfs,
        set_vector_element(Costs3, 3, Cost21, Cost22, Costs4),
        string_element(Cost21, ~(2*Size*(El2+1)+El1), Csf, Cost22),
        V1 := V - Csf,
        update_gain(V1, SecGain, Second, EndSecond, El1, S22, Costs4, Costs5).
otherwise.
```

- We reach the second case if there is no potential gain or if **EndSecond** was reached. Here, we advance the pivot from position **First** until it is at the last position, i.e. **Els** − 2. Note that when we restart the search in the second list, the end of the search becomes **Second**. That means that if the previous case was not executed before **EndSecond**, because no potential gain occured, we don't bother to re-check for potential gains before this position in the next search in the second list.

```
search_perms(Second, _, _, SecGain, S21, Costs1, Costs4) :-
    Second > 0,
    vector_element(Costs1, 6, First),
    vector_element(Costs1, 7, Els),
    First < Els-2 |
        set_vector_element(Costs1, 0, S11, S14, Costs2),
        FNext := First+2,
        set_vector_element(Costs2, 6, _, FNext, Costs3),
        string_element(S11, First, G, S12),
```

```
            V := SecGain+G,
            string_element(S12, FNext, G1, S13),
            V1 := V - G1,
            string_element(S13, ~(FNext+1), Els1, S14),
            search_perms(0, Second, Els1, V1, S21, Costs3, Costs4).
    otherwise.
```

- In the last case, no potential gain could be registered. We transform the incremental gain into a full gain, and we translate positions in the gain lists into relative positions in the sets.

```
    search_perms(_, _, _, SecGain, S21, Costs1, Costs2) :-
        Costs1 = {S11, Size, Cost1, Cost2, P1, P2, First} |
            string_element(S11, First, G, S12),
            V := SecGain+G,
            string_element(S12, ~(P1+1), El1, S13),
            string_element(S21, ~(P2+1), El2, S22),
            Costs2 = {S13, S22, Cost1, Cost2, Size, El1, El2, V}.
```

The following predicate updates the best incremental gain, during the search in the second list.

```
update_gain(V1, SecGain, Second, EndSecond, El1, S2, Costs1, Costs4) :-
    V1 > SecGain, vector_element(Costs1, 6, First) |
        set_vector_element(Costs1, 4, _, First, Costs2),
        set_vector_element(Costs2, 5, _, Second, Costs3),
        search_perms(~(Second+2), EndSecond, El1, V1, S2, Costs3, Costs4).
otherwise.
update_gain(_, SecGain, Second, EndSecond, El1, S2, Costs1, Costs2) :- true |
    search_perms(~(Second+2), EndSecond, El1, SecGain, S2, Costs1, Costs2).
```

The following predicate waits for the result of the permutation search, and calls the correction of the gain lists, which have to account for the new permutation.

```
wait_perms({S11, S21, Cost11, Cost21, Size, El1, El2, G},
           Els, S12, S22, Cost13, Cost23, Per) :-
    string(S11, _, _), string(S21, _, _),
    string(Cost11, _, _), string(Cost21, _, _) |
        correct_gains(~(Els-1), Size, El1, El2, S11, S12, Cost11, Cost12),
        correct_gains(~(Els-1), Size, El2, El1, S21, S22, Cost21, Cost22),
        (wait(Cost12), wait(Cost22) ->
            Cost13 = Cost12, Cost23 = Cost22,
            Per = [G, El1, El2]).
```

The following predicate goes through a gain list and corrects the values of gain according to the permutation. Note that those changes could be restricted to the gains associated to vertices connected to the permuted pair. The cost of the following predicate is minor when compared to the sort, though.
Let's say yet that in an optimized version, we could limit the number of elements to re-sort, if we had such a list. But it would require some structure to store the number of each connected vertex (this is not the global label but the relative position in the set of interest), which would change after each permutation. It would also demand for a structure supporting incremental sort in truly logarithmic time, like a balanced tree.

```
correct_gains(P, Size, El1, El2, S1, S4, Cost1, Cost4) :-
    string_element(S1, P, El) | % implies P > -1
        Pos1 := Size + 2*Size*El,
        string_element(Cost1, ^(Pos1+El1), CostInt, Cost2),
        string_element(Cost2, ^(Pos1+Size+El2), CostExt, Cost3),
        string_element(S1, ^(P-1), Gain, S2),
        set_string_element(S2, ^(P-1), ^(Gain+2*CostInt-2*CostExt), S3),
        correct_gains(^(P-2), Size, El1, El2, S3, S4, Cost3, Cost4).
otherwise.
correct_gains(_, _, _, _, S1, S4, Cost1, Cost4) :- true |
        S1 = S4, Cost1 = Cost4.
```

5 Module test

```
:- module test.
:- public go/5,go2/5,square/2,complete/2.
```

This module has been used to perform tests of the whole program, notably to measure performances. We use the FLIB predicates which are provided for statistic measurement quite extensively here, so refer to [1].

There are two main parts in this module: measurement of performances, for a range of processor numbers and set sizes, and generation of test graphs, namely complete and square graphs.

5.1 Time measurements

In the following predicate, we start the sho-en which will hold time measurements. All time measures are collated in the list `Res` which will be printed at the end of the computation on the console. Parameters of the predicate have the following meaning: `S` is the maximum number of sets, `SM` the minimum, `P` is the maximum number of processors, `PM` is the minimum and `F` is a string holding the name of the data file to read.

```
go(S, SM, P, PM, F) :-
    S >= SM, SM >= 2, P >= PM, PM >= 1, string(F,_,_) |
        fel:get_code(test, go2, 5, Code, normal),
        util:start_stats(Code, {S,SM,P,PM,F}, Res),
        util:sync_wait(Res, Res1),
        util:sync_p_console(Res1, _).
```

The following predicate calls the read of the example file, and measure the time required for this operation.

```
go2(S, SM, P, PM, F) :-
    integer(S), integer(SM), integer(P), integer(PM), string(F,_,_) |
        readfile:read(F, A, B),
        (wait(A), wait(B) ->
            util:req_rel_time(string#"read file", T),
            go3(T, S, S, SM, P, PM, A, B)@priority($,-2000)).
```

Here, after we got the time, we start the Lin Kernighan algorithm, and wait for the result before requesting the time it required to run. The process is iterated with a decreased number of sets, until `SM` sets remain.

```
go3(T, S, MS, SM, P, PM, A, B) :-
    display_console(T), S >= SM, P >= PM |
        util:sync_copy([A,B], [A1,B1], [A2,B2]),
        lk:lk(A1, B1, S, P, R),
        (wait(R), display_console(S), display_console(P) ->
            util:req_rel_time(S, T1),
            go3(T1, ~(S/2), MS, SM, P, PM, A2, B2)).
otherwise.
go3(T, S, MS, SM, P, PM, A, B) :-
    S < SM |
        go3(T, MS, MS, SM, ~(P-1), PM, A, B).
otherwise.
go3(_, _, _, _, _, _, _, _) :- true | true.
```

5.2 Usage

Let us give some clues about the usage of this module for measurements of the performance of the algorithm: with and without cheating. Results are presented in section 6 but let's make some remarks here.

- What happens we the number of sets grows, for a given graph?
  More and more dummy nodes are increased. That's a way to measure acceleration due to parallelism when the cost of the Lin-Kernighan algorithm is minimal, i.e for a single pass involving sets with one vertex only. When the number of precessors increases, the intermediary structure built on each processor is used for less and less pairs of sets, until only one pair is computed through the Lin-Kernighan procedure on each processor.
  In this case, the cost of building the intermediary structure grows like s/p, where s is the number of sets, which is supposed to be much larger than the number of vertices; the cost of communication grows like $s \log_2(p)$, and the cost of each Lin-Kernighan run is nearly constant, while on a given processors, $s/2/p$ pairs have to be processed. All in all, s pair exchanges have to be done, and we can assume that only 2 passes are done, for s large enough.
  In summary, complexity of the algorithm should be near:

$$\mathcal{O}\left(s^2(\alpha/p + \beta \log_2(p))\right)$$

  for large s's.

- For a given graph with n vertices, and a much smaller number of sets, complexity figure is quite different. Communication cost grows like $n \log_2(p)$, while Lin-Kernighan runs cost $\mathcal{O}\left(\frac{n^2}{s \times p} \log_2(n/s)\right)$. Number of passes depends on the graph complexity and on the number of sets, but is usually limited to 4 or 5. It is independent of p, in any case. Number of pair exchanges is still s. In summary, complexity of the algorithm should be near:

$$\mathcal{O}\left(\alpha \frac{n^2}{p} \log_2(n/s) + \beta \times s \times n \log_2(p)\right)$$

The results of both types of measurements are done in section 6. It's noteworthy that we don't need anything but a trivial graph to perform the first measurement, which gives us an idea of the worst case gain we can expect in terms of speed up. This is what we called cheating.

In order not to cheat, the predicate described in the following section can be used to generate more complicated graphs.

5.3 Generation of a complete graph

A complete graph is a graph in which each vertex is connected with all other vertices in the graph. In our case, cost is unit cost, for all edges. The following predicate opens file in write mode, write the number of vertices and call the graph generation predicate, for a complete graph of size `N`. Generation is done at a lower priority than file opening to avoid piling the write request in memory. This causes more suspensions to occur but makes execution more smooth.

```
complete(N, File) :-
    N > 0, string(File, _, _) |
        fel:open_file(Stream, File, w, Status),
        (Status = normal ->
            Stream = [putt(N), putc(#"."), nl | Stream1],
            complete(0, N, Stream1)@priority($,-2000);
         otherwise;
         true ->
            util:p_console([string#"Can't open file:", File], _),
            Stream = []).
```

The following predicate deals with the generation of edges for the vertex with label L, amongst the N vertices.

```
complete(L, N, Stream) :-
    L < N |
        Stream = [putt(L), putc(#"."), nl, putt(~(N-1)), putc(#"."), nl
                  | Stream1],
        complete(L, 0, N, Stream1, Stream2, Sync),
        (wait(Sync) -> complete(~(L+1), N, Stream2)).
otherwise.
complete(N, N, Stream) :- true | Stream = [].
```

Eventually the following predicate generates all edges for a vertex. Note that the reflexive edge is not generated.

```
complete(E, E, N, Stream, Stream2, Sync) :- true |
        complete(E, ~(E+1), N, Stream, Stream2, Sync).
otherwise.
complete(L, E, N, Stream, Stream2, Sync) :-
    E < N |
        Stream = [putt(E), putc(#"."), nl, putt(1), putc(#"."), nl | Stream1],
        complete(L, ~(E+1), N, Stream1, Stream2, Sync).
otherwise.
complete(_, _, _, Stream, Stream2, Sync) :- true | Stream = Stream2, Sync = 1.
```

5.4 Generating of a square graph

A sqare graph is a graph in which each vertex is connected to each of its 4 neighbors, when applicable, i.e. besides edges. All edges have unit cost. The following predicate opens file in write mode, write the number of vertices and call the graph generation predicate, for a square graph of size N^2. Generation is done at a lower priority than file opening to avoid piling the write request in memory. This causes more suspensions to occur but makes execution more smooth.

```
square(N, File) :-
    N > 1, string(File, _, _) |
        fel:open_file(Stream, File, w, Status),
        (Status = normal ->
            Stream = [putt(~(N*N)), putc(#"."), nl | Stream1],
            square(0, 0, N, Stream1)@priority($,-2000);
         otherwise;
         true ->
            util:p_console([string#"Can't open file:",File],_),
            Stream = []).
```

The following predicate deals with the generation of edges for the vertex with label $Y \times N + X$, amongst the N^2 vertices.

```
square(0, 0, N, Stream) :- true |
        Stream = [putt(0), putc(#"."), nl, putt(2), putc(#"."), nl,
                  putt(1), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(N), putc(#"."), nl, putt(1), putc(#"."), nl | Stream1],
        square(1, 0, N, Stream1).
square(0, Y, N, Stream) :-
```

```
    Y > 0, Y < N-1 |
        L := Y*N,
        Stream = [putt(L), putc(#"."), nl, putt(3), putc(#"."), nl,
                  putt(~(L+1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L-N)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+N)), putc(#"."), nl, putt(1), putc(#"."), nl
                  | Stream1],
            square(1, Y, N, Stream1).
square(0, Y, N, Stream) :-
    Y =:= N-1 |
        L := Y*N,
        Stream = [putt(L), putc(#"."), nl, putt(2), putc(#"."), nl,
                  putt(~(L+1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L-N)), putc(#"."), nl, putt(1), putc(#"."), nl
                  | Stream1],
        square(1, Y, N, Stream1).
square(X, 0, N, Stream) :-
    X > 0, X < N-1 |
        L := X,
        Stream = [putt(L), putc(#"."), nl, putt(3), putc(#"."), nl,
                  putt(~(L-1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+N)), putc(#"."), nl, putt(1), putc(#"."), nl
                  | Stream1],
        square(~(X+1), 0, N, Stream1).
square(X, Y, N, Stream) :-
    X > 0, X < N-1, Y > 0, Y < N-1 |
        L := X+Y*N,
        Stream = [putt(L), putc(#"."), nl, putt(4), putc(#"."), nl,
                  putt(~(L-1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L-N)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+N)), putc(#"."), nl, putt(1), putc(#"."), nl
                  | Stream1],
        square(~(X+1), Y, N, Stream1).
square(X, Y, N, Stream) :-
    X > 0, X < N-1, Y =:= N-1 |
        L := X+Y*N,
        Stream = [putt(L), putc(#"."), nl, putt(3), putc(#"."), nl,
                  putt(~(L-1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L-N)), putc(#"."), nl, putt(1), putc(#"."), nl
                  | Stream1],
        square(~(X+1), Y, N, Stream1).
square(X, 0, N, Stream) :-
    X =:= N-1 |
        L := X,
        Stream = [putt(L), putc(#"."), nl, putt(2), putc(#"."), nl,
                  putt(~(L-1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+N)), putc(#"."), nl, putt(1), putc(#"."), nl
                  | Stream1],
        square(0, 1, N, Stream1).
```

```
square(X, Y, N, Stream) :-
    X =:= N-1, Y > 0, Y < N-1 |
        L := X+Y*N,
        Stream = [putt(L), putc(#"."), nl, putt(3), putc(#"."), nl,
                  putt(~(L-1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L-N)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L+N)), putc(#"."), nl, putt(1), putc(#"."), nl
                  | Stream1],
        square(0, ~(Y+1), N, Stream1).
square(X, Y, N, Stream) :-
    X =:= N-1, Y =:= N-1 |
        L := X+Y*N,
        Stream = [putt(L), putc(#"."), nl, putt(2), putc(#"."), nl,
                  putt(~(L-1)), putc(#"."), nl, putt(1), putc(#"."), nl,
                  putt(~(L-N)), putc(#"."), nl, putt(1), putc(#"."), nl].
```

6 Examples

This section has two points: first, to illustrate the usage of the Lin-Kernighan procedure with the partitioning result for "small" examples. Then, to show how execution speed was increased by our parallel implementation.

6.1 Simple examples

We show here some the result of partitioning a grid-shaped graph, as produced by the module `test`. In figure 1, we show the result for a 10x10 graph, for a 4-way partition.

```
2 2 2 1 1 1 0 0 0 0
1 2 2 1 1 1 0 0 0 0
2 2 2 1 1 1 1 0 0 0
1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 0 0 2
0 0 0 3 3 2 2 2 2 2
0 0 0 3 3 2 2 2 2 2
0 0 0 3 3 3 3 3 2 2
3 3 3 3 3 3 3 3 2 2
3 3 3 3 3 3 3 3 2 2
```

Figure 1: 4-way partitioning of a 10x10 grid

In figure 2, we show the result for a 20x20 graph, still for a 4-way partition.

```
0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 3 3 3 3 3
0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 3 3 3 3 3
0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 3 3 3 3 3
0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 3 3 3 3 3
0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 3 3 3 3 3
0 1 1 1 1 1 1 1 0 0 0 0 0 2 2 3 3 3 3 3
0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3
```

Figure 2: 4-way partitioning of a 20x20 grid

Eventually, in figure 3, we show the result for a 32x32 graph, this time for a 32-way partition.

Let's note that for a reasonable case of standard-cell partitioning, assuming we use one set per row, we can expect that a 128-way partitioning of a graph with 10000 vertices will be required. Average degree of

```
22 22 27 23 16 16 16 23 23 23 18 29 29 29 29 29 29 06 06 06
22 22 22 23 23 23 16 16 25 25 25 18 18 29 29 29 29 29 29 29
22 22 22 22 22 28 28 28 28 25 25 18 18 18 18 18 18 30 30 30
22 11 11 11 11 28 28 28 28 28 18 18 18 31 31 31 31 31 30 30
09 09 20 11 11 11 28 28 28 17 17 17 17 10 10 31 31 31 30 30
09 09 09 04 20 20 20 20 20 17 17 17 17 10 10 10 10 10 30 30
09 09 04 04 04 23 23 23 23 17 00 17 17 17 10 10 10 21 21 21
09 09 09 04 04 23 23 16 03 17 00 00 00 08 08 08 08 08 08 19
12 12 09 15 04 04 04 04 03 03 05 00 00 00 20 15 15 08 19 19
12 09 09 15 04 04 04 03 03 03 19 00 00 00 20 15 20 19 19 19
03 03 06 06 06 16 16 16 03 03 19 14 00 00 00 25 25 25 19 19
26 06 06 06 06 16 16 03 03 03 14 14 14 25 25 25 25 25 19 19
26 06 06 06 11 11 16 16 14 14 14 24 24 24 24 24 20 20 20 19
26 26 26 02 02 02 31 31 14 14 14 24 24 24 24 24 20 01 01 01
26 26 26 26 02 02 02 02 14 14 11 24 07 07 24 24 01 01 01 01
26 26 26 26 02 02 02 02 10 10 10 07 07 07 15 15 15 01 01 01
30 30 30 21 02 01 01 01 07 07 07 07 07 07 15 15 15 05 12 05
12 21 21 21 21 21 27 27 27 27 13 13 13 13 15 15 08 05 05 05
12 12 12 21 21 21 27 27 27 27 13 13 13 13 08 08 08 05 05 05
12 12 12 31 31 21 27 27 27 27 13 13 22 13 13 13 08 05 05 05
```

Figure 3: 32-way partitioning of a 30x30 grid

a vertex in the graph is near 6, in most practical cases, whereas in the case of a grid, degree of a vertex is close to 4.
Those figures suggest a basic time complexity 50% higher than for a grid-shaped graph with the same number of vertices. For a 128-way partitioning, intrinsic speed-up factor is limited to 64. This figure can be improved if we further increase the number of sets by assigning several sets to one row. That will also improve the quality of the subsequent set-to-row assignement. This would be the basis for a bottom-up hierarchical cell-placement strategy.

6.2 Graphs with no edges

To use graphs with no edges to measure performances of the Lin-Kernighan algorithm seems curious at the least. Our point here is to measure precisely the cost of features which are not pertaining to the pure Lin-Kernighan processing. In our case, we have to consider the cost of broadcasting to all processors the string holding current partition and the cost of building an intermediary structure at each pass of the algorithm instead of only modifying the current structure.

Now, let's remark that during the course of our program, we add "dummy" nodes to the graph so that the number of vertices is a multiple of the number of sets of the partition. Thus, if we give to our program a small graph and a large number of sets, many such "dummy" nodes will be created, and the optimal partition of a graph will be looked for, such that there is one vertex per set. Very clearly, no optimization is necessary to achieve this, so that the Lin-Kernighan algorithm will converge immediatly. Therefore, in order to measure the cost of parallel artifacts, we took a graph with only 1 vertex, and we changed the number of sets s used in the partition.

A rough expression of the complexity of those operations is given in section 5. Our task is now to verify that the hypothesis upon which we based those expressions conform to measurements.

Figure 4 shows the computing time for a range of processors, as a function of the number of sets. We can check that the complexity actually grows like the square of s.

Figure 5 shows the coefficient of the curves above as a function of the number of used processors. We see that complexity, as the number of processor increases, decreases like the inverse of the number of processors.

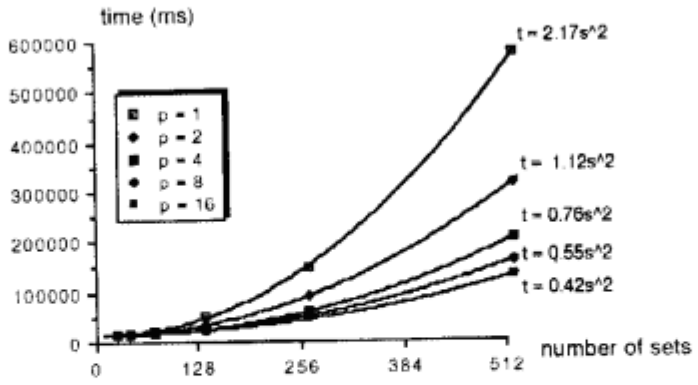

Figure 4: Computation time versus the number of sets

It is difficult to measure the communication cost of the implementation, because it is a second order factor in the cost, and the coefficients of figure 4 are first order approximations.

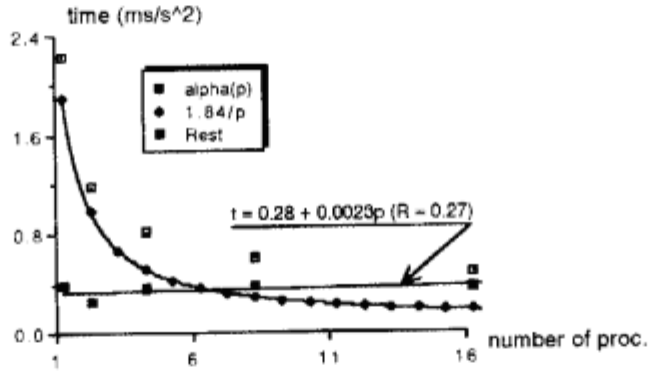

Figure 5: Time factor versus the number of processors

Therefore, we have to analyze a specific case in order to measure the communication cost. In figure 6, we show the computing time, for a 32-way partition, as a function of the number of processors. We see that this time is the sum of an hyperbolic function and a linear function. The latter equals the hyperbolic function when $p \sim 8$. It means we can expect a speed up of 8 in the worst case, i.e. when no optimization is necessary. This figure, theoritically, does not depend on s.

As a matter of fact, when the graph size increases, we shall see that this figure improves, as predicted by the formula of section 5. Let's note also that the linear curve of figure 6 is unexpected, and let us think that implementation (especially synchronization) is not correct. We could not check this further as the 64-processors machine is currently down, but we noticed at the least that speed still increases for more than 8 processors, when s grows larger and larger.

6.3 Grid-shaped graph

A grid shaped graph can give us more realistic figures of the complexity of the algorithm. Several runs have been done, for a 32-way partition of a n by n grid, with n chosen amongst 10, 20, 30 and 40.

Figure 7 pictures execution time of the partitioning program for those grids, as the number of processors increases. Whereas dots correspond to actual figures, curves are an approximation of the logarithmic functions which best matches experimental points. Expression in regard of each curve indicates that as the

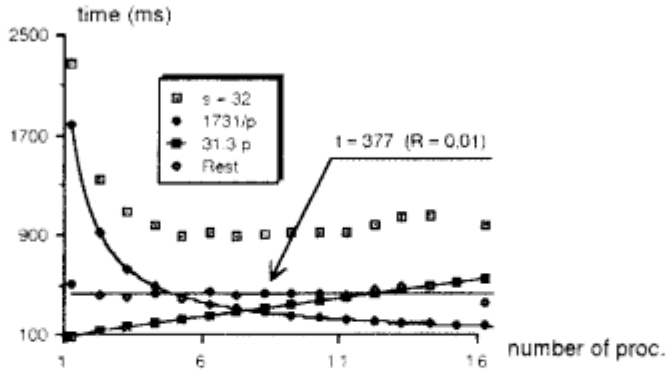

Figure 6: Computation time versus the number of processors ($s = 32$)

number of vertices increases, speed up figure gets closer to a linear function of p.

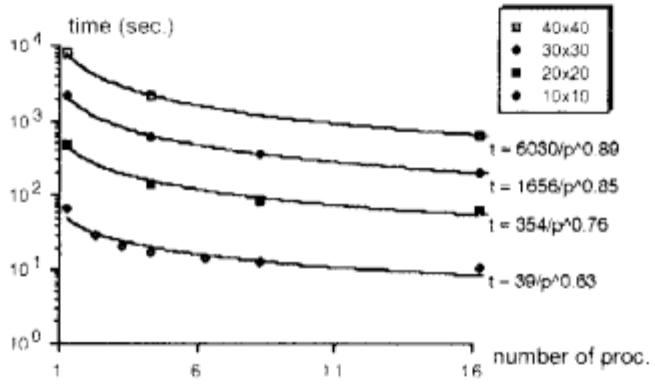

Figure 7: Computation time versus the number of processors for grid-shaped graphs

Accurate speed up figures can be found in table 1

p	10x10	20x20	30x30	40x40
4	3.7	3.5	3.5	3.6
8	5.2	6	5.6	
16	7.3	10.6	11.7	

Table 1: Speed up of partitioning, for 1 to 16 processors, and various grids sizes

These figures, although satisfying are a little bit far from the linear speed-up we would expect: communication cost and construction of the intermediary structure does not take a long time and divides well over several processors, as shown in section 6.2.

There are two main reasons, according to us, to explain the slow-down:

1. Some computations may be more heavy on some processors, due to the non uniformity of the problem introduced by the initial placement and the arbitrary order in pair processing.

2. When less than $s/2$ processors are used, pairs are allocated statically to processors. This may further accentuate problem 1.

To further check that no artifact has been introduced, which could explain the slow down, let's see how the partioning speed evolves with the number of nodes in the graph, from figure 8.

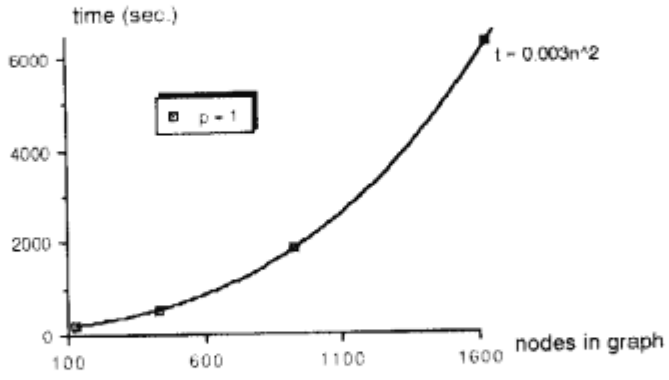

Figure 8: Computation time versus the number of nodes, for a single processor

We can see that complexity evolves quadratically with the number of nodes. The logarithmic factor does not appear. It means that the heuristic is efficient, and that a nearly constant number of trials has to be done to find optimal permutations between two sets.

Eventually, let's note that an optimized sequential version may be faster than our version running on a single processor. For the matter of asymptotic complexity though, figure 8 proves that the dominant factor is not due to the creation of intermediary structures (the main point to be optimized on a sequential version) but to the determination of the optimal permutation.

6.4 Conclusions

We could register encouraging performance results, for middle-sized graphs. The cost of parallelism over the sequential version can be decomposed into three main components:

- the communication cost, when current partition is broadcasted, after each pass, and when permutations are collated together: this accounts for a negligible franction of the computing time.
- the parallel implementation cost, which is due to the creation of an intermediary data structure (parti cost matrix) on each processor for each pass: this account for a small part of the total computing time.
- the static ordering of pairs chosen for optimization: this is the major problem of our implementation, causing a speedup plateau to appear.

The solution of this problem lies maybe in dynamic allocation of pairs to processors. We would like to do that, but the main problem is to find a continuous sequence of pairs, (i_k, j_k) such that $\forall u, j,\ 0 \leq i < j \leq s$, $\exists k \in [1, s(s-1)/2]$, such that $i = i_k$ and $j = j_k$, and $|k - k'| < p \Rightarrow i_k \neq i_{k'} \wedge j_k \neq j_{k'}$.

Contents

References

[1] B. Burg and D. Dure. Flib user manual. Technical report, Icot, 1989.

[2] B. W. Kernighan and S. Lin. An efficient heuristic procedire for partitioning graphs. *Bell System Technical Journal*, 49:291–308, February 1970.