Design, Implementation and Evaluation of **ParaDict**, a Data Parallel Library for Dictionaries

Joaquim Gabarró[*] Jordi Petit i Silvestre[*]
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Campus Nord — Mòdul C6
Jordi Girona Salgado, 1–3
08034 Barcelona, Spain
{gabarro,jpetit}@lsi.upc.es

February 24, 1997

Abstract

ParaDict, a data parallel library for dictionaries having two different interfaces is presented. The first interface is written in C* for data parallel users and the second interface in C, for users that want to use a parallel library but are not willing to write parallel programs. An inneficient prototype implementation (using associative memories) and an efficient implementation (using 2-3 trees) are presented. We have seen that C* is an adequate tool to code theoretical PRAM algorithms into readable programs. These programs were ran on a CM 200 with better times than other existing implementations. Morover, they also have much better asymptotic behaviour when compared to a sequential implementation on a workstation. Finally, the relationship between data parallelism and vectorization is explored, transforming C* code into C code plus compiler directives and running the result on a Convex C3480 machine.

Keywords: Parallel libraries, dictionaries, 2–3 trees, data parallelism, design implementation and evaluation methods, C*, vectorization techniques.

[*]This work has been partially supported by ESPRIT LTR Project no. 20244 — ALCOM-IT.

1 Introduction

Sequential abstract data types are well known from theory and practice and complete libraries exist for them. For instance, the well known LEDA library developed by K. Melhorn and S. Näher [16] written in C++ is widely used (both in industry and in academia). However, the situation is quite different on massive parallelism. We have a very sophisticated theory on parallel data structures (cf. J. JáJá [14]) but very few practical work. Therefore, it was really tempting to explore the "practical" issues of this theory writing (a small part of) a library in a commercial data parallel language and running it on a real data parallel machine.

In this paper, we consider the design, implementation and evaluation of a data parallel library for handling dictionaries (ParaDict). A dictionary is an abstract data type (ADT) which stores keys associated with informations. The main operations available are *looking up* if a given key belongs to the dictionary, *accessing* the information of a given key, *inserting* a key along with its associated information, and *deleting* a key. The interface of ParaDict has been done carefully, in order to offer these operations in a usefull set of data types, functions and procedures, which can be used from parallel and sequential environments. The parallel interface is written in C* and generalizes the one presented in LEDA's dictionaries; the sequential interface offers the same operations than the former one, but the public functions are written in C.

In order to have a first workable version of the library for parallel dictionaries and enabling the testing of its interface, a prototype has been designed using associative memories. Its encoding and evaluation on a Connection Machine with 2K processors emphasize its inefficiency. The final version of the library has been accomplished coding the algorithms on 2-3 trees given by W. Paul, U. Vishkin and H. Wagener in [18]. Since these algorithms were originially described in an informal style, our task has been done in two steps. First, we wrote the algorithms in a pidgin data parallel language close to the one given by D. Hillis and G. Steele in [12]. Widely applying stepwise refinement development techniques we obtained clear, complete and readable high level algorithms. In the second step, we coded these algorithms into the C* programming language [24]. To evaluate the resulting code, the running time of some usual operations has been measured on a Connection Machine with 16K processors [13].

The use of complex data structures over vectorial computers was considered porting a subset of the library to a CONVEX vectorial machine citeConvexC. To do it, we show a way to transform parallel algorithms coded in C* to sequential programs coded in C along with compilation directives. We call this set of transformations $C^{\#}$ (informally $C^{\#} = C +$ `#pragmas`). These programs are efficiently executed in this computer thanks to the vectorizing optimizer. Some experimental measures allow us to compare the parallel and vectorial dictionaries.

Let us quickly review some other related works. Parallel processing of data structures is a theoretically well known domain: taking only the research based on PRAMs we have, among others, the work of W. Paul, U. Vishkin and H. Wagener based on 2-3 trees [18] (on which our implementation is based), the work of L. Higham and E. Schenks on B-trees [11], the work of J. Gabarró, C. Martínez and X. Messeguer based on Skip Lists [9] or the work of J. Gabarró and X. Messeguer based on AVL trees [10]. However, less implementations have been realized. We know the works done by M. Gastaldo *et al.* [6], by X. Messeguer [17] and J.L. Träff [25]. We shall compare their results with ParaDict.

Reasoning about data parallel programs is an vital activity. Designing our programs we follow the approach given by J. Gabarró and R. Gavaldà in [8]. The study of the program correctness in the data parallel case is now a well established topic, consider for instance the work done by L. Bougé *et al.* [3] or the approach taken by A. Stewart in [22].

Regarding the implementation of parallel libraries, there are also some other ingoing projects. For instance, the PAD library [15] will offer basic parallel algorithms and data structures for the SB-PRAM. This library includes dictionaries among other data structures. Finally, considering real machines and aiming to users without expertise in parallelism, Frames [23] will provide support for the programming of distributed memory machines via libraries of "programming frames".

The remaining of this paper is organised as follows. First of all, we exhibit three different kinds of library models we have found usefull, and develop their use by the means of a toy example. Then we show how, inspired by the LEDA sequential library, we have designed ParaDict's interfaces. After that, we describe part of our implementation, first using a prototype and then using parallel algorithms on 2-3 trees. We present some experimental results aiming to evaluate it and to compare ParaDict to other implementations. Finally we show how we have ported the C* programs to a vectorial machine. The paper is closed with some concluding remarks.

2 Data parallel libraries

Three library models. Let us describe three models of programming and using libraries that correspond to different environments. We schematize them in Figure 2. The first one is the classical library model, widely used in sequential environments:

SP/SL model: A Sequential Program calls a Sequential Library. In this case, the program and the library are both written in a sequential language, let's say, C. This is the so-called *pure sequential* approach.

The second model is a generalization of the preceeding one to data parallel languages:

PP/PL model: A Parallel Program calls a Parallel Library. This approach is addressed to data parallel programmers. This means in practice a C* program calling functions written in C*. This is the *pure parallel* approach.

Our last model appairs when a sequential program uses operations contained in the previous parallel library:

SP/PL model: A Sequential Program calls a Parallel Library. Addressed to sequential programmers. Since C* contains C as a subset, programmers can write C code and call operations of a parallel library. This library starts transforming sequential data into parallel data and runs data parallel procedures in the parallel system. Due to this transformation, a *bridge level* between the sequential program and the parallel library must be inserted. Depending on the problems this approach can be interesting. We call this model the *mixed* approach.

A toy example. Let us develop a little bit more these three models with a toy example based on sorting integers. In the SP/SL model the sequential program can use the procedure

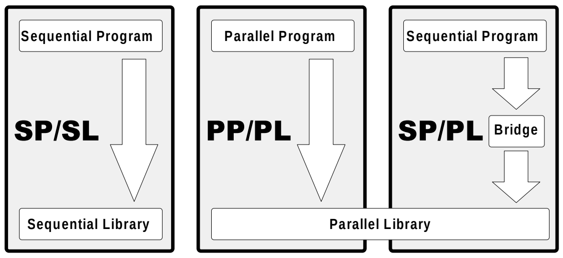

Figure 1: Library models.

```
void SPSL_Sort (int a[], int n) {
    Sequential sorting code
}
```

In this case, the sequential program would call `SPSL_Sort(a,n)` where `a` is a sequential array. For the PP/PL model, the main data parallel program would call `PPPL_Sort(&a)` where `a` is a parallel variable having a shape decided at run-time (using the `current` keyword) in order to have a *neutral data distribution* [7]:

```
void PPPL_Sort (int:current *a) {
    [rank(*a,0,CMC_upward,CMC_none,CMC_no_fill)]*a=*a;
}
```

this procedure sorts the parallel variable `*a` by applying the `rank` function, provided by the C* standard comunination library.

In the SP/PL approach, the sequential programmer calls `SPPL_Sort(a,n)` where `a` is a sequential array variable:

```
void SPPL_Sort (int a[], int n) {
    shape [n]s;
    with (s) everywhere {
        int:current pa;
        pa=write_to_pvar(a);    /* FE->CM */
        PPPL_Sort(&pa);
        read_from_pvar(a,pa);   /* CM->FE */
}   }
```

this procedure uses the C* primitives `write_to_pvar` and `read_from_pvar` connecting sequential and parallel variables. To construct the SP/PL library from the PP/PL case, we just needed to add a *bridge* level (see Figure 1) made of two steps that wrap the parallel program: the first (`write_to_pvar`) spreads the sequential array to the parallel subsystem, the second (`read_from_pvar`) gathers the result back to the front end. Between them, the pure parallel sorting function can be invoked. Remark that SP/SL and SP/PL headers (`Sort(int a[],int n)`) coincide.

Of course, it can be argued that the introduction of the bridge level can produce a decrease of the performances of programs, mainly due to the overload of connecting parallel and sequential variables. Figure 2 shows the time needed to communicate sequential and parallel variables in our system. Despite of this overload, we have found that this approach is still useful for some kinds of problems. For instance, Figure 3 compares the running time of the previous sorting programs. Thus, it can be seen which is exactly the fraction of time consumed by the bridge and that (even with it) the SP/PL model is faster than the SP/SL (which uses `qsort` from `stdlib.h`). This will happen, in general, when dealing with big problems, with large amounts of data or long execution times.

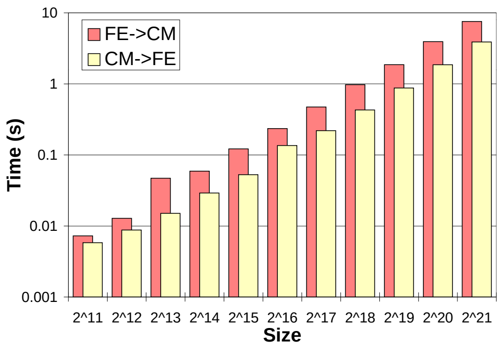

Figure 2: Measured time to transfer arrays of integers between the front-end (FE) and the Connection Machine (CM) on a CM 200 with 2K processors.

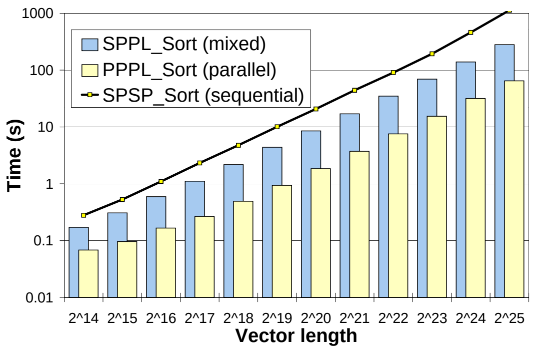

Figure 3: Running time for sorting vectors in dependence of the model: SP/SL (Sun Sparc 10), SP/PL and PP/PL (both with a CM 200 with 16K processors).

3 The design of ParaDict's library interface

The library we present, ParaDict, implements the operations to efficiently handle dictionaries on parallel computers using a data parallel approach. Concretely, we offer two different implementations (a prototype kind, and a 2-3 tree based kind) with two different interfaces: a parallel one and a sequential one. The former interface is written in C* and it is aimed for parallel programs (PP/PL model); the latter is written in C and it is designed to be used within sequential programs (SP/PL model). Both interfaces contain the same operations which are based on the sequential implementation of dictionaries in LEDA [16]. We think that this is a good starting point, due to the relevance of the LEDA library in sequential computing. Refer to [21] for a complete description of ParaDict's interfaces.

Extending operations towards parallelism. In order to use parallelism, LEDA headers have to be enhaced. The way to do it in C* is easy: where LEDA expects a single key, information or item, ParaDict expects a parallel variable of them (ie, a set). In C*, a parallel variable can be seen as a usual array where each component has a (virtual) processor associated to it. For instance, let us consider the operation that, given a key in a dictionary returns its associated information. The name of this method in LEDA is `Access` and its header in C++ syntax is

```
TInf d.Access (TKey key)
```

where `TKey` is the type of the keys, `TInf` is the type of the informations and `d` is a dictionary object. The C* header for the same operation in the PP/PL interface of ParaDict is

```
TInf:current Access (TDict *d, TKey:current keys)
```

where `TDict` is now the dictionary ADT. The meaning for the parallel version is the extension of the sequential one: for each active key, its associated information in the dictionary is returned. Following the conventions of C*, all our functions maintain the meaning of the context, so the inactive positions will not be treated.

The main functions that form the library parallel interface are shown along with their documentation in Figure 4. The hole PP/PL interface is given in Appendix A.1. The types included in these descriptions are the following:

- `bool` is a C* standard type representing a boolean. The values it can hold are 0 or 1.
- `TDicc` is the dictionary abstract data type defined by the library.
- `TKey` is the type of the keys to be included in the dictionary.
- `TInf` is the type of the informations to be included in the dictionary.
- `TItem` is the item abstract data type defined by the library. Items are the same abstraction of pointers than in LEDA.

It should be remembered that only active components in the current context are processed.

Constructing the sequential interface. In the sequential interface, the same operations than in the parallel interface are available. However, in order to follow ANSI C, some of the parameters have been changed and some have been added. In fact, since all these functions work with open arrays, the user has to supply an integer `k` representing their size. A new parameter called `mask` simulating the setting of the context in C* has also been added (for convenience, passing `NULL` as its value will enable the hole context). Thus, the C header of the SP/PL version of the `Access` procedure is

```
void Access_ (TDict *d, int k, bool mask[], TKey keys[], TInf infs[])
```

We give the hole SP/SL interface of ParaDict in Appendix A.2. The meaning of the functions is the same than in the PP/PL.

```
TItem:current Insert (TDict *d, TKey:current keys, TInf:current infs)
```

Effect: Inserts every key in `keys` with associate information in `infs`.
Returns: The items where the keys were inserted.
Note: If some key was already present in `d`, its information is modified according to the new information passed as argument.
Note: When requesting to add several instances of the same key (possibly with different informations), the information stored in the corresponding item is chosen arbitrarily by the implementation among the ones received.

```
TItem:current Lookup (TDict *d, TKey:current keys)
```

Returns: A collection of items with keys in `keys`. If no such item exist, the item returned is **nil**. The "nilness" of items can be tested with the `IsNil` function.

```
void DelItems (TDict *d, TItem:current items)
```

Effect: Deletes from `d` every item in `items`.
Precondition: Every item in `items` is present in `d`.
Note: Some instances of the same item may be removed.

```
void Change (TDict *d, TItem:current items, TInf:current infs)
```

Effect: Changes the information field stored in every item in `items` by `infs`.
Precondition: Every item in `items` is present in `d`.

```
bool:current IsNil (TDict *d, TItem:current items)
```

Returns: `true` if item is **nil**.

```
TKey:current Keys (TDict *d, TItem:current items)
```

Returns: The key of every active item in `items`.
Precondition: Every active item in `items` is in `d`.

```
TInf:current Infs (TDict *d, TItem:current items)
```

Returns: The information of every active item in `items`.
Precondition: Every active item in `items` is in `d`.

Figure 4: Main functions of ParaDict's parallel interface and their description.

Lacks and drawbacks. The most important of them is the lack of genericity or polymorphism. Once the types `TKey` and `TInf` have been defined, dictionaries parametrized with them have to be used. Another important restriction is that, for the current implementations, the `TKey` type has to be arithmetic. It is hard to correct these drawbacks in a coherent form without changing the language in which the code of library is written, C*. A more object oriented, yet data parallel language could be of interest.

4 Prototyping ParaDict

Sometimes it is not clear if theoretically good and sophisticated algorithms will run in practice better than poor but easy to implement alternatives. Before developing an efficient version of ParaDict using elaborated data structures, we made a prototype. The focus was not aimed to obtain an efficient implementation but, instead, a first version quick to develop that could enable us to test the PP/PL and SP/PL interfaces. In order to get a reusable design we faced at these three points:

- In the PP/PL interface, headers accept data in any format, but algorithms use more restrictive formats. Therefore a *preprocessing step* is necessary.
- The *bridge level* in the SP/PL approach can be fully developed with the prototype.
- Experimenting with the prototype we can detect where *optimizations* are needed.

Preprocessing step at the PP/PL level. A large number of times, theoretical algorithms assume some preconditions that clearly cannot be accepted by a library for general use. For instance, parallel insertion algorithms always assume that the keys to be inserted are not previously contained in the tree, that they are sorted in an array in contiguous positions from 0 to $n-1$, and that there are no repeated keys [18, 11, 9]. As a consequence, since ParaDict wants to free the user from enforcing these annoying preconditions, it must apply some kind of *preprocessing* to the input.

Annalizing the common preconditions of the algorithms, we developed wrapers for the functions that preprocess the input in order to adapt them to the more restrictive formats. Going on with the insertion algorithm, we wrote the next function that 1) searches the keys already present and changes its values; 2) sorts and collapses the rest of the keys to contiguous positions of an array; 3) removes the repeated keys and collapses again the array; 4) calls `Insert2` where the actual insertion algorithm is applied; and 5) unmakes all these transformations to return the result in the same positions that the user requested.

```
TItem:current Insert (TDict *dict, TKey:current keys, TInf:current infs) {
    TItem:current items,items2;
    int:current r1,r2;
    int n1,n2;

    items=Lookup(dict,keys);
    where (items!=NIL) {
        Change(dict,items,infs);
    } else {
        n1=Count(); r1=Rank(keys); [r1]keys=keys; [r1]infs=infs;
        everywhere where (Ord() < n1) {
            where (Ord()==0 || [.]keys!=[.-1]keys) {
             n2=Count(); r2=Enum(); [r1]keys=keys; [r1]infs=infs;
             where (Ord() < n2) {
```

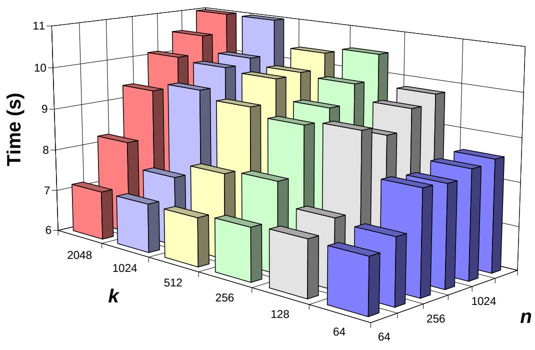

Figure 5: The measured times to insert n keys and search k on the prototype motivate the search of more involved algorithms.

```
                items2=Insert2(dict,keys,infs,n2);
           }   }
           items2=[r2]items2;
       } items=[r1]items2;
   } return items;
}
```

This strategy has two advantages: it allows us to deal in a clean way with data preprocessing and it offers us a clear interface to deal with the future optimizations, which must reside inside the `Insert2` function.

Bridge at the SP/PL model. Since we want to offer a sequential interface to out parallel library, we have to devellop a bridge. The bridge concept has been explained in Section 2 and one of its advantages is that when the headers are fixed, it is independent of the actual implementation. Thus, we could write and test the SP/PL level for the first prototype knowing that it would not change when developing an efficient version of the library. This establishes the modularity of the approach. The details can be found in [20].

Prototype implementation and evaluation. The prototype implementation of ParaDict was made with arrays as main data structure. This gave us straightforward algorithms, easy to understand and quick to implement. The preprocecessing of the inputs and the bridge level were developed for this level (and later reused in the efficient version of ParaDict without changes). This prototype was tested on a small Connection Machine with 2K processors. Due to the small size of memory in this machine and the naivity of the subjacent algorithms, the capacity of the dictionary was very small (only a few thousands keys) and the execution times very bad (as expected). Figure 4 reports the time needed to insert n keys and search k keys in function of n and k.

Of course, these magnitudes are largely higher than what can be expected from parallel computers, and thus, motivate the use of more involved algorithms and data structures.

5 2-3 trees: from informal descriptions to C* programs

To implement an efficient version of ParaDict we choose the algorithms given by W. Paul, U. Vishkin and H. Wagener [18, 14] based on 2-3 trees (a class of trees where all leaves have the same depth and internal nodes have two or three sons). These EREW PRAM algorithms have time $\mathcal{O}(\log n + \log k)$ and need k processors, where k is the number of keys to search (insert or delete) and n is the number of leaves in the tree (the size of the dictionary).

We percieved 2-3 trees interesting because they are an irregular structure and because the algorithms involve many interesting programming points. For instance, they all use *divide and conquer*, the parallel search for a batch of keys is based on synchronous *packet routing* and the parallel insertion and deletion procedures use a bottom-up tree recontruction using *pipelines*. We found that programming all these techniques was not obvious and quite challenging for a C* programmer. As a consequence, the implementation of the algorithms was done in two steps.

- In the first step, we transforme the informal algorihms into a not ambiguous description, somehow related to the data parallel pidging language defined by Hillis and Steele [12]. We follow the directions given by Gabarró and Gavaldà [8] dealing with the correctness of data parallel algorithms and to apply the same modularity and descending design techniques well-known in the case of designing sequential algorithms. As in [9], the result was a clear and readable highlevel description of the original informal algorithms.

- In a second step, we convert this pseudocode to the C* data parallel language. It was then not a so tremendous task and, against our first impression, we found C* well adapted.

Moreover, these two steps were done using literate programming, ie joining the source code and its descriptive textual documentation in a single document. The complete documented design and implementation of ParaDict can be found in [19]. Due to space reasons, in this section we only try to give a flavour of it by describing some operations and data structures.

5.1 Data structure representation

In the following, 'Key' is used to design the key type and that 'Inf' designs the type of the associated informations. A node of the tree has the following representation:

```
record Node is                                                   (1)
    parent                :↑ Node
    Lptr, Mptr, Rptr      :↑ Node
    Lkey, Mkey, Rkey      : Key
    key                   : Key
    inf                   : Inf
    urn                   : natural
    pos, sons             : {0, 1, 2, 3}
end record
```

The interpretation is the same as in the sequential case [1]. Redundant fields exist, according to the original algorithms in [18]. A new field *urn* has been added, its use will be apparent latter. A 2-3 tree is a '23Tree' record:

```
record 23Tree is                                                 (2)
    root :  ↑ Node
end record
```

Let us briefly show how the preceeding definitions have been implemented in C*. A cell corresponds to a node. The `Lk,Mk,Rk,Lp,Mp,Rp` and `Pos` fields correspond, respectively, to the *Lkey*, *Mkey*, *Rkey*, *Lptr*, *Mptr*, *Rptr* and *pos* fields defined in (1). The cells contain a `free` boolean flag in order to enable garbage collection.

⟨*Cell type*⟩≡
```
typedef struct {
    bool    free;
    TItem   parent;
    TKey    Lk,Mk,Rk;
    TItem   Lp,Mp,Rp;
    TKey    key;
    TInf    inf;
    char    sons,pos;
} TCell;
```

In the final implementation the data types `TItem` and `TDict` must be also defined. The 2-3 data type given in (2) is transformed in the dictionary data type `TDict`. Its most important field is `cells`, a pointer to a parallel variable of cells where nodes are stored. We also include a pointer to the root of the tree (`root`) and a pointer to the last occupied cell (`last`). The other informations that are always maintained are the height of the tree (`height`) and its size (`size`).

⟨*Dictionary type*⟩≡
```
shape [MaxCells]SCells;

typedef struct {
    TItem           root;
    int             height;
    TItem           last;
    nat             size;
    TCell:SCells    *cells;
} TDict;
```

The `TItem` type is simply an integer used as a pointer to a node (as a $\uparrow$ Node) that indexes over the `TDict.cells` array. We use the special value `-1` using the `NIL` constant to designate **nil**.

⟨*Item type*⟩≡
```
typedef int TItem;
#define NIL (-1)
```

5.2 A search algorithm with pack routing

The search algorithm follows the *packets routing* tecnnique introduced by Paul, Vishkin and Wagener for 2-3 trees [18] (also used by Highan and Shenk for B-trees [11] and by Gabarró, Martínez and Messeguer for Skip-lists [9]). At the start of the algorithm, there is a unique active packet located at the root of the tree. This packet contains all the keys to search, sorted by keys. At each stage, each active packet is descended by routing it to the left, middle or right of the node on which it is located, or is splited into two new packets because the hole packet cannot entirely descend. When a packet reaches a leaf, it becomes inactive. The algorithm ends when all packets are inactive. At most, k processors are needed to execute this loop, one for each active packet. Moreover, the decision of routing a packet to one of the three directions or to split it needs a constant time, independently of the number of the keys inside the packet. This approach does not produce concurrent reads (it is and EREW algorithm).

Overview of the highlevel algorithm. The basic data structure we need (besides the tree) is the 'Packet'. Each key *keys*[*i*] to search, belongs to a packet *pckts*[*j*]. A packet has the following structure:

```
record Packet is                                                        (3)
    $firstKey, lastKey$ :   Key
    $state$ :               $\{Passive, Active, Located\}$
    $first, last$ :         integer
    $nodeP$ :               $\uparrow$ Node
    $nodeC$ :               Node
end record
```

The fields *first* and *last* are pointers to the first and last key of a packet. *firstKey* and *lastKey* are those keys. The field *nodeP* is a pointer to the node where the packet is located and *nodeC* is a copy of it (it contains the keys and pointers to the sons and parent). These redundances will avoid concurrent reads. The function *Locate* creates a set *pckts* with a unique active packet that contains all the keys to search (*InitPackets*) and, from the root to the leaves, splits and routes down this set of packets (*RoutePacket*). When all the packets become inactive, the located leaves are notified to each member of every packet.

```
function Locate                                                         (4)
    in $T$ : 23Tree
    in $keys$ : array $[0 .. k-1]$ of Key
    returns $nodes$ : array $[0 .. k-1]$ of $\uparrow$ Node is

    preconditions
        $k > 0, \quad Size(T) = n > 0, \quad \forall i : 0 < i < k : keys[i-1] < keys[i]$
    var
        $pckts$ : array $[0 .. k-1]$ of Packet

    begin
        $InitPackets(T, pckts, keys)$
        — Main loop —
        while $\exists i \in [0 .. k-1] : pckts[i].state = Active$ do
            for all $i \in [0 .. k-1]$ such that $pckts[i].state = Active$ do in parallel
                $RoutePacket(T, i, pckts, keys)$
        — Spreading of the found leaves to members in a packet —
        for all $i \in [0 .. k-1]$ such that $pckts[i].state = Located$ do in parallel
            for all $j \in [pckts[i].first .. pckts[i].last]$ do in parallel
                $nodes[j] := pckts[i].nodeP$
    end Locate
```

In procedure *RoutePacket*, *Direction* returns the values *left*, *middle*, *right* or *split* allowing the packets to move down one step (to the left, right or middle son), to split if they collide with a key in the tree and, finally, to stop when they arrive to a leaf. *RoutePacket* routes packet *pckts*[*i*] according to its direction.

```
procedure RoutePacket                                                   (5)
    in $T$ : 23Tree
    in $i$ : integer
    in/out $pckts$ : array $[0 .. k-1]$ of Packet
    in $keys$ : array $[0 .. k-1]$ of Key is

    begin
    case $Direction(T, pckts[i])$ of
        $left$ $\longrightarrow$    $pckts[i].nodeP := pckts[i].nodeC.Lptr$
                      $pckts[i].nodeC := pckts[i].nodeC.Lptr\uparrow$
        $middle$ $\longrightarrow$  $pckts[i].nodeP := pckts[i].nodeC.Mptr$
                      $pckts[i].nodeC := pckts[i].nodeC.Mptr\uparrow$
        $right$ $\longrightarrow$   $pckts[i].nodeP := pckts[i].nodeC.Rptr$
                      $pckts[i].nodeC := pckts[i].nodeC.Rptr\uparrow$
        $stop$ $\longrightarrow$    $pckts[i].state := Located$
        $split$ $\longrightarrow$   $SplitPacket(T, i, pckts, keys)$
    end case
end $RoutePacket$
```

C* implementation. Let us now considerer the implementation of the previous algorithms in C*. The packet type of algorithm (3) is now specified as the following structure record:

⟨*Packet type*⟩≡
```
typedef struct {
    TKey  firstK,lastK;
    nat   firstP,lastP;
    char  state;
    TItem node; /* nodeP */
    TCell cell; /* nodeC */
} TPacket;
```

The *Locate* function in (4) is written in C* in the following chunk of code. Note that now a segmented scan is used to spread the found leaves. Remark also, that the `Lookup` function developed for the prototype can directly call this fuction.

⟨*Locate function*⟩≡
```
TItem:current Locate (TDict *d, TKey:current keys) {
    TPacket:current pckts;
    bool:current b;

    InitPackets(d,&pckts,keys);
    while (|= (pckts.state==Active))
        where (pckts.state==Active)
            RoutePackets (d,&pckts,keys);
    b=pckts.state==Located;
    return scan(pckts.node,0,CMC_combiner_copy,CMC_upward,
                CMC_start_bit, &b ,CMC_inclusive);
}
```

Finally, algorithm (5) becomes the code:

⟨*RoutePackets procedure*⟩≡
```
void RoutePackets (TDict *d, TPacket:current *pckts, TKey:current keys) {
    char:current dir=Direction(*pckts);

    where (dir==DirL) {
        pckts->node=pckts->cell.Lp;
        pckts->cell=[pckts->cell.Lp]*d->cells;
    } else where (dir==DirM) {
        pckts->node=pckts->cell.Mp;
        pckts->cell=[pckts->cell.Mp]*d->cells;
    } else where (dir==DirR) {
        pckts->node=pckts->cell.Rp;
```

```
            pckts->cell=[pckts->cell.Rp]*d->cells;
        } else where (dir==Stop) {
            pckts->state=Located;
        } else /* where (dir==Split) */ {
            SplitPackets(d,pckts,keys);
    }   }
```

We would like to emphatize that C* code mimics descriptions given in the pidgin data parallel language. Therefore our two step design proved to be a very good strategy because it allowed us to deal with increasing complexity levels in an adequate way.

Direct CREW parallelization. We have implemented another search procedure based in a simple parallelization of the sequential case [20]. Since the search of a key does not modify the tree, searching a set of keys can be done by different processors at the same time, each of them independently taking care of one key. The disadvantage of this approach is that a machine with concurrent read capabilities is needed. We have found that in the Connection Machine, for a sufficiently big numbers of keys to search, this is slightly slower than the first one.

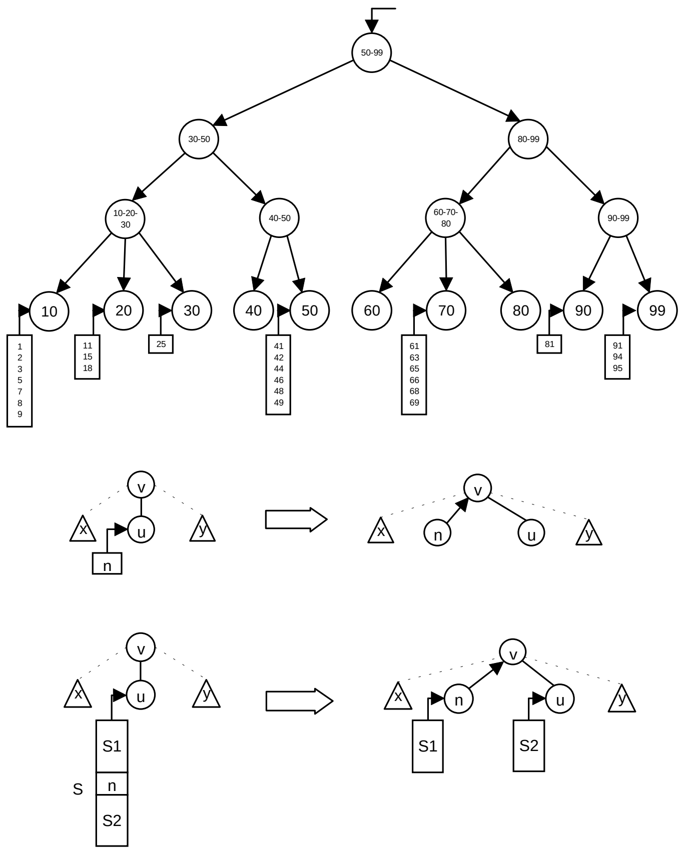

Figure 6: Insertion start up: (a) The set of groups (b) evolution of a group with exactly one key, (c) the general case, a divide and conquer strategy.

5.3 A pipelined insertion algorithm

In the following, we describe the algorithm which basically follows the ideas given in [18, 14]. We assume that the keys to insert are not in the tree, and that none of them is greater than the greater key stored in the tree.

First of all, the algorithm searches the keys to insert. Since these keys are not stored in the tree, the returned nodes by *Locate* correspond to their insertion point. Furthermore, since all the keys are less than the greater key stored in the tree, all of them are at the left of their insertion point. A *group* is the set of all keys with the same insertion point. For new key-information pairs, new leaves are allocated. This situations corresponds to the part (a) in Figure 6. For instance, keys 11, 15, 18 have the same insertion point, therefore they form a group hanged to 20. The core of the algorithm begins now and there are two cases depending on the number of keys in the grups.

Each group contains exactly one key. The procedure to insert k new leaves is a bottom-up process of the tree, by levels, as in the sequential case [1]. However, in the parallel case, more than one node has to be treated at every level. This case is illustrated in the part (b) of Figure 6 where only the packet n has been made explicit. At the beginning, each new group, for instance $n = leaves[i]$, to be inserted is hanged to the father v of its insertion point u. Since v was an internal node, it had two or three sons. After this parallel hanging, the node v has between three and six sons. The nodes with three sons remain unchanged. The nodes v with more than three sons split. Node v creates a new brother v' and the sons of v are distributed among v and v'. The process is now repeated for the next level, hanging all new nodes v' to the fathers of v. When reaching the root, it can be necessary to do a root-up operation, ie to increment the height of the tree. Figure B given in Appendix B, explains the splitting process in a more detailed way.

General case, where each group can have more than one leaf to hang. This case corresponds to the part (b) in the Figure 6. Let $S = \{leaves[f], \ldots, leaves[l]\}$ be the packet containing the (ordered by keys) set keys having u as insertion point. Now we apply the previous algorithm for the middle elements n of each group, ie to the elements $leaves[s]$ where $s = (f + l)$ **div** 2. The group S has now to be divided in two new smaller groups $S_1 = \{leaves[f], \ldots, leaves[s-1]\}$ and $S_2 = \{leaves[s+1], \ldots, leaves[l]\}$ where the insertion point of S_1 is n and the insertion point of S_2 remains being u. Now the preceding case applies with n and u hanged of v and a pipeline can start again. This *divide and conquer* process ends when all the groups are empty.

The final algorithm consists of intercalating the two previous procedures. The first one is called *HangsUp* and the second *LeavesUp*. We can observe that *LeavesUp* does not need to wait for all the tree to be processed before hanging news leaves. It only has to wait until the hanging requests are sufficiently high. Concretely, applying this *pipeline*, we only have to wait that the two last levels of the tree are ready before launching new leaves. As a consequence, the final algorithm treats the tree at different levels at the same time, with a prudential distance between them in order not creating interferences. Thereby, the final algorithm alternates two executions of *HangsUp* with one of *LeavesUp*. The insertion ends when there are no more groups with leaves to insert and when all the hanging requests have been served.

High level algorithms. The basic data is the 'Hang' record, which contains the hanging requests and guides the pipeline. A hang request is described by its state (*act*) and the source node *src* that is to be hanged to the destination node *dst* from position *pos* w.r.t. the sons of *dst*.

```
record Hang is                                                    (6)
    act :    boolean
    src :    ↑ Node
    dst :    ↑ Node
    pos :    $\{0, 1, 2, 3\}$
end record
```

Procedure *Insert* does the preprocessing:

```
procedure Insert                                                  (7)
    in/out T : 23Tree
    in keys : array [0 .. $k-1$] of Key
    in infs : array [0 .. $k-1$] of Inf is

    preconditions
        $k > 0$,   $Size(T) = n > 0$,   $\exists key \in T : keys[k-1] < key$
        $\forall i : 0 < i < k : keys[i-1] < keys[i]$,   $\forall i : 0 \leq i < k : keys[i] \notin T$
    var
        nodes, leaves : array [0 .. $k-1$] of ↑ Node
    begin
        nodes := Locate(t, keys)
        leaves := CreateLeaves(t, keys, infs)
        Insertion(T, leaves, nodes)
    end Insert
```

Procedure *Insertion* receives a sorted array with k new leaves (*leaves*) to insert into T and their corresponding insertion points (*nodes*). A hanging request is associated with each new leaf, as well as a boolean indicating if it is the head of a group (*heads*) and a number meaning at which distance is located the next group's head (*dists*). At initialization, all hanging requests are made inactive and the other variables are setup according to their definition. The main loop intercalates one call to *LeavesUp* with two calls to *HangsUp*. The *LeavesUp* call introduces new requests to the pipeline; the *HangsUp* call advances them one stage in the pipeline. In order to ensure that the successives stages of the pipeline are separated at least by one level of the tree, two calls of *HangsUp* must be made before calling *LeavesUp*. The loop ends when no hanging request are still active.

```
procedure Insertion                                               (8)
    in/out T : 23Tree
    in leaves : array [0 .. $k-1$] of ↑ Node
    in nodes : array [0 .. $k-1$] of ↑ Node is

    var
        hangs :   array [0 .. $k-1$] of Hang
        heads :   array [0 .. $k-1$] of boolean
        dists :   array [0 .. $k-1$] of natural
    begin
        — Initialize —
        for all $i \in [0 .. k-1]$ do in parallel
            $hangs[i].act$ = false
            $heads[i] := i = 0 \vee_c nodes[i-1] \neq nodes[i]$
        end for
        dists := Reverse Segmented Enumeratarion on heads
        — Main loop —
        $LeavesUp(leaves, nodes, heads, dists, hangs)$
        while $\exists i \in [0 .. k-1] : hangs[i].act$ do
            $HangsUp(T, hangs)$
            $HangsUp(T, hangs)$
            $LeavesUp(leaves, nodes, heads, dists, hangs)$
        end while
    end Insertion
```

A complete description of the functions *LeavesUp* and *LeavesUp* can be found in Appendix B. The preceding approach allows us to deal clearly with a pipeline. Note that no especial constructs are necessary, moreover this approach can be clearly translated to C* code, as we can see bellow.

C* implentation. The `THang` type identifier defined in the following structure corresponds to record 'Hang' of algorithm (6).

⟨*Types for inserting*⟩≡
```
typedef struct {
    bool act;
    TItem src,dst;
    char pos;
} THang;
```

The following `Insertion` program corresponds to procedure *Insertion* of algorithm (8):

⟨*Insertion*⟩≡
```
void Insertion (TDict *d, TItem:current nodes, TItem:current leaves) {
    THang:current hangs;
    bool:current heads;
    nat:current dists;

    heads=Ord()==0 || [.-1]nodes!=[.]nodes;
    dists=enumerate(0,CMC_downward,CMC_inclusive,CMC_segment_bit,&heads)-1;
    hangs.act=false;

    LeavesUp (d,leaves,&nodes,&heads,&dists,&hangs);
    while (|= (hangs.act)) {
        HangsUp(d,&hangs);
        HangsUp(d,&hangs);
        LeavesUp (d,leaves,&nodes,&heads,&dists,&hangs);
}   }
```

5.4 Pipelined deletion algorithm

The deletion algorithm works in a similar way to the insertion one: a `TEraser` auxiliary data structure is used to direct the divide and conquer and the pipeline processes. The C* structure is

⟨*Types for the deletion*⟩≡
```
typedef struct {
    char state;
    TItem source;
} TEraser;
```

and the main deletion function is

⟨*Deletion*⟩≡
```
      void Deletion (TDict *d, TItem:current items) {
          TEraser:current erasers;

          InitErasers(&erasers,items);
          while (|= (erasers.state!=Passive) {
              RemoveLeaves(d,&erasers);
              ArrangeLeaves(d,&erasers);
              ArrangeLeaves(d,&erasers);
      }   }
```

6 Experimental results and comparisons with other implementations

In order to evaluate the performance of some usual operations of our library, we have measured and analyzed their running time on a CM 200. Experiments have been repeated enough times; results shown below are their mean. The variances were not substancial, as the measures were done using the machine in exclusive mode.

Evaluation of the Lookup and Insert operations. The experimental results obtained for searching or inserting k keys in a dictionary storing n elements are shown in figure 6. For comparison with a well-known workstation, we also show the times needed for the equivalent sequential insertions. We conclude that, with our machines, even if the sequential implementation is faster than the parallel one for reasonable values of k, the time increase is smoother, making clear the scalability of our parallel library.

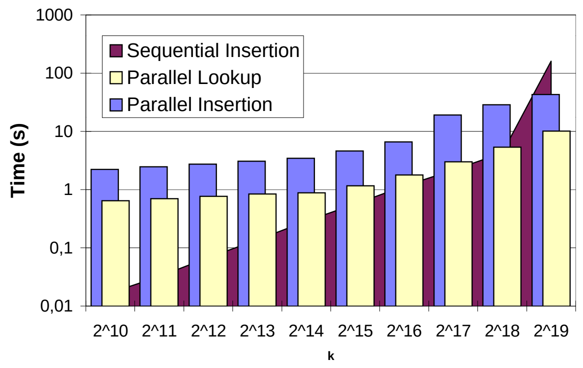

Figure 7: Running time for searching / inserting k keys in a dictionary of size n=150000 on a CM 200 with 16K processors. In the background, insertions on a Sun Sparc 10.

Comparisons with other data structures. Figure 6 compares results from [17] with ours. Since the experimental conditions where the same, we can affirm that ParaDict's implementation with 2-3 trees is slightly more efficient than X. Messeguer's implementation based on skip lists. Moreover, it saves space and can store much more elements.

The comparison of our results against the ones given in [6] by M. Gastaldo, shows that our implementation is 5 times faster, Figure 6. For instance, an insertion of 500000 elements can be done in 43 seconds (7 if the dictionary is empty) on a Connection Machine with 16K processors with ParaDict, whereas on a MasPar-1 with 1K processors it takes 240 seconds. However, we have to cautious about this kind of information, because we are not only comparing the algorithms but the parallel machines involved in the measurements.

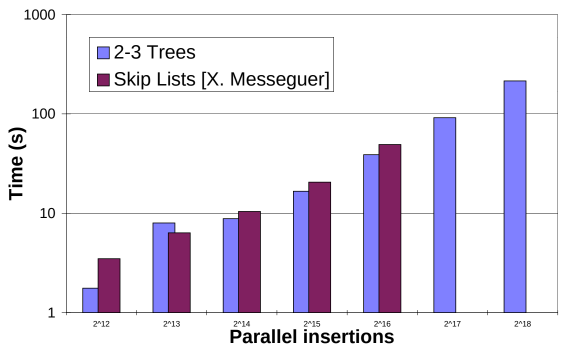

Figure 8: Running times for inserting k elements in a dictionary of size n=150000 when using 2-3 trees (ParaDict) or skip lists [17] on a CM 200 with 2K processors.

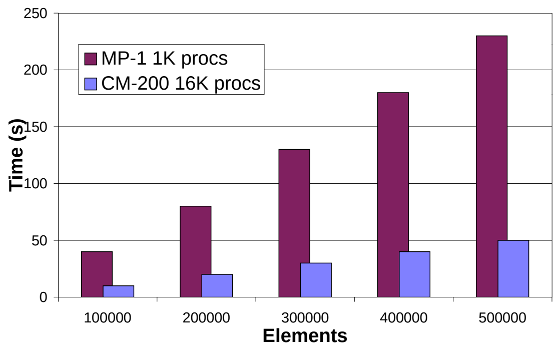

Figure 9: Insertion times in a MasPar-1 with 1K processors according to [6] and in a Connection Machine with 16K using ParaDict.

Comparison with the PAD library. PAD [15] is a general parallel library written in FORK [15] (a parallel extension of C) that among other data structures offers dictionaries. PAD does not yet run on any real parallel machine but has been tested using a PRAM simulator. The matter is that the authors give results measured in SB-PRAM clock cycles. In Figure 6 we reproduce one of their experiments: a dictionary is built with 3000 items, 128 keys are inserted and deleted, 256 news keys are inserted and deleted and the dictionary is finally destroyed. As the time scales and machines can not be compared, we cannot

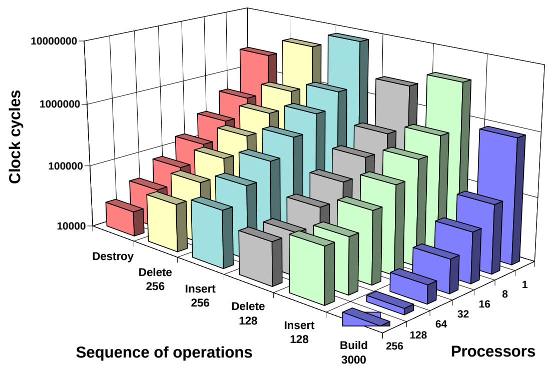

Figure 10: Time (in SB-PRAM clock cycles) to build a dictionary with 3000 items, insert 128 keys, delete them, insert 256 keys, delete them and destroy the dictionary in function of the number of processors. Results come from the parallel implementation of J.L. Träff simulated on a workstation [25].

say which implementation is "better", but some differences can be stated: 1) ParaDict is a data parallel library, whereas PAD is assynchronous. 2) ParaDict has been designed with fine grain parallelism whereas PAD is coarse grain. This can only not be seen in the internal implementations but also in the number of keys that can are handled in parallel: with ParaDict we are used to insert concurrently batches of 2^{18} keys, whereas PAD's authors insert at most 256 keys in parallel. 3) Results in PAD have been simulated on a sequential machine, however ParaDict has been tested in a CM 200 with 16K processors. It will be interesting to have information about the preceding PAD experiments on any real machine, because the hole approach seems promissing.

7 C$^{\#}$, vectorization using data parallelism

There is a lot of resemblance between the data parallel and vectorial programming models: both consist of applying the same operation to different data. For this reason, we found interesting to port our parallel programs written in C* to a vectorial computer (a Convex C3480, a machine with registers of 128 elements and 8 banks of memory). We achieved it by defining a set of transformations to convert C* code to C code augmented by compiler directives (which have the `#pragma _CNX` form) [5]. In the following, this language is called C$^{\#}$. In this paper, we describe only some of these transformations. The complete set we used can be found in [20].

Send operation. The first transformation we show is for the *send* operation, which in C* is written as:

```
int:current dst,src,indx;
[indx]dst=src;
```

where `indx` is supposed to be a permutation. Its correct transformation to a C$^{\#}$ efficient program is the following:

```
int src[],dst[],indx[];
for (i=0; i<n; i++) indx2[i]=indx[i];
for (i=0; i<n; i++) src2[i]=src[i];
#pragma _CNX force_vector
for (i=0; i<n; i++) dst[indx2[i]]=src2[i];
```

Auxiliary copies have to be generated, because these vectors could be the same. The compiler directive has to be given, since the compiler cannot recognize `indx2` as a permutation. This operation (or its dual, the *get* operation) can be efficiently executed if the processor has *scatter* (or *gather*) instructions, which is the case on the Convex.

Enumeration. The enumeration operation is very common in data parallel programming. For instance, it can be used to identify each processor in a range $[0..n-1]$. In C* this operations is made via the `enumerate` primitive:

```
int:current a;
a=enumerate(0,CMC_downward,CMC_exclusive,CMC_none,CMC_no_field);
```

Its transformation in C$^{\#}$ is straightforward:

```
int a[];
for (i=0; i<n; i++) a[i]=i;
```

and the compiler does not have any problem to fully vectorize it.

Parallel prefix. This operation is related to another important operation in data parallel programming. Given an operator $\oplus$ and a vector $a = [a_0, \ldots, a_{n-1}]$, we define $/\!/_{\oplus} a = [b_0, \ldots, b_{n-1}]$ where $b_i = \bigoplus_{j=0}^{i} a_j$. In C* this function is called `scan` and belongs to the standard communication library. A naive implementation would be:

```
for (b[0]=a[0], i=1; i<n; i++)
    b[i]=b[i-1]+a[i];
```

but the vectorizing compiler could not solve the recurrence and would leave this loop scalar. However, implementing the classic PRAM algorithm [14] as:

```
for (i=0; i<n; i++) b[i]=a[i];
for (j=1,p=pow2(j-1); j<=log2(n); j++) {
  for (i=p/2; i<n; i++) aux[i]=b[i];
  for (i=p; i<n; i++) b[i]=aux[i-p]+aux[i];
}
```

the compiler can stripmine the `i` loops. We have found that on the Convex machine, the second implementation is 2.5 times faster than the first one when $n > 4096$. This result is negative, since according to Amdahl's law, this low speedup will have effect on all the algorithms that contain it.

Conditioning. Let us now consider the conditioning instruction `where (cond) S`. We code the active context with an unidimentional array. As the contexts have a block structure, we need a stack. This approach has been already considered by L. Bougé and J. Levaire [4] to give an operational semantics to a basic data parallel language, $\mathcal{L}$. The corresponding transformation is:

```
for (i=0; i<n; i++) if (cond[i]) S(i);
```

The instruction `where (cond) S1 else S2` can be rewritten as `t=cond; where (t) S1; where (!t) S2`, and the preceding technique applies.

As the Convex C3480 and CM 200 machines have a similar masking behaviour the transformation is efficient. Both have also the same drawback: the execution time does not decrease when only few unmasked elements of a vector are processed.

Segmented scans. Segmented scans have been extensively studied by G. Blelloch in [**?**, 2]. Assume we have an array v and another array f of flags. Each flag specifies the start of a new segment. For instance, if we consider the following array $v = [1\ 1\ 1\ 7\ 1\ 4\ 1\ 5\ 6\ 8\ 4]$ and $f = [1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0]$ the segmented array is represented as $s = [1\ 1\ 1 \mid 7\ 1 \mid 4\ 1\ 5\ 6 \mid 8\ 4]$ and the segmented prefix sum will be $/\!\!/_{+}s = [1\ 2\ 3 \mid 7\ 8 \mid 4\ 5\ 10\ 16 \mid 8\ 12]$. The segmented version of an $\oplus$ operation can be implemented as $(v_a, f_a) \otimes (v_b, f_b) = (v_r, f_r)$ where $v_r = v_b$ if f_b holds, $v_r = v_a \oplus v_b$ if $\neg f_b$ holds and $f_r = f_a \vee f_b$. In C$^{\#}$ we get the following code:

```
for (j=1,k=log2(n); j<=k; j++) {
    p=pow2(j-1);
    for (i=p/2; i<n; i++) {
        v2[i]=v[i]; f2[i]=f[i];
    }
    for (i=p; i<n; i++) if (!f2[i]) {
        v[i]=v2[i-p]+v2[i];
        f[i]=f2[i-p];
}   }
```

Example: Radix sort. Sorting is also a usual operation on data parallel machines. Radix sort is a good candidate because it is easily implemented on the CM 200 machine and it will be interesting to see what kind of C vectorizable code we will obtain. First of all, recall the `RadixSort` procedure written in C* as:

```
void RadixSort (nat:current *a, nat n) {
    nat:current enu;
    nat d,k,D=boolsizeof(nat:current);

    for (d=0,k=1; d<D; d++,k<<=1) {
where (*a & k) {
            enu=n-1-enumerate(0,CMC_downward,CMC_exclusive,CMC_none,CMC_no_field);
        else
            enu=enumerate(0,CMC_upward,CMC_exclusive,CMC_none,CMC_no_field);
}
     [enu]*a=*a;
}   }
```

Its transformation yelds the following:

```
void RadixSort (nat a[], nat n) {
    nat enu[MAX],aux[MAX];
    nat i,d,k,e,D=sizeof(int)*8;

    for (d=0,k=1; d<D; d++,k<<=1) {
        for (e=n-1,i=n-1; i>=0; i--) if (a[i]&k) enu[i]=e--;
        for (e=0,i=0; i<n; i++) if (!(a[i]&k)) enu[i]=e++;
        for (i=0; i<n; i++) aux[i]=a[i];
        for (i=0; i<n; i++) a[enu[i]]=aux[i];
}   }
```

Note that the transformation into $C^{\#}$ give us a code where only the more external loop remains sequential. The other loops corresponding to enumerations, copy and send operations have been vectorized by the Convex C compiler.

A first judgement. We presented a way to transform *directly* data parallel programs written in C* into $C^{\#}$ programs. Therefore it is possible to transform data parallelism into highly vectorizable code and we have a connection between these two approaches to parallelism. However, as we have seen in the parallel prefix transformation, the speedup is not so important. Therefore, transformations seem to be good but the speedup seems to be bad. To get a better inside into the behaviour of $C^{\#}$ we developed a vectorial version of ParaDict. Next section explores the results.

8 $C^{\#}$ vectorial dictionaries

Applying these kinds of transformations to a subset of ParaDict, we have obtained its vectorial implementation (with the SP/PL interface). Even if almost every loop was made vectorial by the optimizing compiler, the performances achieved at run-time were very poor: table 8 reports it. The speedup of the vectorial implementation with respect to the sequential one is 5 for the building operation and 2 for the search operation. Vectorizing the insertion operation is self-defeating.

Operation	CM 16K	Convex	
(Measures in seconds)	Par	Sca	Vec
Build with 2^{20} leaves	7.118	26.723	7.658
Search 2^{16} keys	1.776	6.624	3.340
Insert 2^{14} keys	—	2.407	40.309

Table 1: Some measures characterizing the behaviour of the different implementations. Searches and insertions are made on a tree with 150000 leaves.

We conjecture that the reason for these modest improvements (when they exist) is again the highly irregular structure we are dealing with, and the bottleneck it creates accessing the limited set of memory banks.

9 Conclusions

In parallelism, there seems to be a gap between theory and practice. In many cases it is quite difficult to measure the effort necessary to transform an informal algorithm into readable code. We got a pleasant surprise with C*, because sophisticated algorithms were easily coded. As in the sequential case, the program development by stepwise refinement has been extensively used to get readable programs.

Moreover it has been possible to define two complete and useful interfaces: a sequential and a parallel one. This is important because they reflect two different views of parallelism.

The sequential interface is planned to be used by sequential programmers using data parallelism in a hidden way. The data parallel interface is to be used by programmers having a knowledge of data parallelism. Both classes of programmers coexists today. People coming from computer science uses friendly data parallel environments, but most of the people coming from other disciplines prefers a sequential environment. However both classes of programmers can take advantatge of data parallel libraries. More important, only one kind of data parallel program has to be developped. These parallel programs can be used from a sequential environment just adding a *bridge level*, always easy to build. We claim that the whole approach will remain true for other applications.

Experimental results left open many questions. Whereas our dictionary implementation (a highly irregular and dynamic object) performs better than previous implementations, parallel results are not so good in relation to their sequential counterpart. However, in many cases these comparisons are not so clear and, in our case, involve rather old SIMD machines against new and powerful sequential computers. It is unclear what happens with more modern parallel machines. Moreover, in the near future, compilers for data parallel languages on MIMD machines could become more popular. If this happens, data parallel programming could become every day programming, but care has to be taken when dealing with irregular data structures as ours.

Many questions remain open about the distinction between *high* and *low level* approaches to programming in parallel. Programming with a sequential languages plus a message passing library (e.g. PVM) is for us a low level approach. We also consider a low level activity to write (directly) good sequential vectorizable programs. From the other side we consider C* programs as high level. Transformations from C* to $C^{\#}$ connects high and low levels, but the results were poor. We guess the same will happen if we try a transformation from data parallel programs to sequential programs + message passing. In both cases the problem seems to be the *fine grain* and the *irregularity* of the application, but this is just a feeling without any theoretical proof. In any case, we think that data parallel languages will continue to be an interesting and elegant high level counterpart to other low level approaches.

Further information

The source files of our code, the experimental data obtained in our runs and further information regarding ParaDict can be found on the World Wide Web at the address

http://www-lsi.upc.es/~jpetit/ParaDict

Acknowledgements

The implementation and evaluation of our programs has been done at the Centre Europeu de Paral·lelisme de Barcelona (CEPBA, now $\mathcal{C}^4$) and at the Parallelldatorcentrum (PDC) of the Kungl Tekniska Högskolan (Sweden). We acknowledge X. Messeguer for help and letting us use his code.

References

[1] A. Aho, J. Hopcroft, and J. Ulman. *The Desing and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[2] G. Blelloch. *Vector Models for Data-Parallel Computing.* MIT, 1990.

[3] L. Bougé, D. Cachera, Y. Le Guyadec an d G. Utard, and B. Virot. Formal validation of data-parallel programs: A two-component assertional proof system for a simple language. In A. Darte and G.-R. Perrin, editors, *The Data-Parallel Programming Model*, volume 1132 of *LNCS Tutorial*, pages 252–281, June 1996. Invited conference.

[4] L. Bougé and L. Levaire. Control structures for data-parallel SIMD language: Semantics and implementation. *Future Generation Computer Systems*, (8):363–378, 1992.

[5] Convex Computer Corporation. *Convex C Optimization Guide*, 1991.

[6] Th. Duboux, A. Ferreira, and M. Gastaldo. MIMD dictionary machines from theory to practice. In *Conpar92-VappV*, LNCS 634, pages 545–550. Springer-Verlag, 1992. Look also M. Gastaldo PhD thesis: Contribution à l'algorithmique parallèle des structures de données discrètes: machines dictionnaire et algorithmes pour les graphes, ENS-Lyon, 1993.

[7] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.

[8] J. Gabarró and R. Gavaldà. An approach to correcness of data parallel algorithms. *JPDC*, (22):185–201, 1994.

[9] J. Gabarró, C. Martínez, and X. Messeguer. A design of a parallel dictionary using skip lists. *TCS*, (158):1–33, 1996.

[10] J. Gabarró and X. Messeguer. Massively parallel and distributed dictionaries on AVL and Brother trees. In *Parallel and Distributed Computing Systems*, pages 14–17. ISCA, 1996.

[11] L. Higham and E. Schenks. Maintaining B-trees on an EREW PRAM. *JPDC*, (22):329–335, 1994.

[12] D. Hillis and G. Steele. Data parallel algorithms. *CACM*, 29(12):1170–1183, 1986.

[13] W. Daniel Hillis. *The Connection Machine.* MIT, 1987.

[14] J. JáJá. *An Introduction to Parallel Algoritms.* Addison Wesley, 1992.

[15] C. Keßler and J.L. Träff. Language and library support for practical PRAM programming. In *5 Euromico Workshop on Parallel and Distributed Processing*, pages 216–221. IEEE, 1997.

[16] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *CACM*, 38(1):96–102, 1995.

[17] X. Messeguer. A sequential and parallel implementation of skip lists. Technical Report LSI-94-41-R, LSI-UPC, November 1994.

[18] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3-trees. In *10th ICALP*, LNCS 154, pages 397–609. Springer-Verlag, 1983. Look also: Parallel computation on 2-3-trees, RAIRO, 397–404, 1984.

[19] J. Petit. ParaDict. A data parallel implementation of dictionaries with 2-3 trees. Technical report, LSI-UPC, September 1996. LSI-96-7-T.

[20] J. Petit. Llibreria de diccionaris paral·lels: disseny, implementació, avaluació, January 1996. FIB-UPC Computing Project.

[21] J. Petit. ParaDict's User Guide. Technical report, LSI-UPC, May 1996. LSI-96-4-T.

[22] A. Stewart. Reasoning about data-parallel array assignment. *JPDC*, (27):79–85, 1995.

[23] T. Römke and J. Petit. Programming frames for the efficient use of parallel systems. Technical report, LSI-UPC, January 1997. LSI-97-9-R. Also appeared as TR-001-97, Paderborn Center for Parallel Computing.

[24] Thinking Machines Corporation. *C* Programming Guide*, 1991.

[25] J.L. Träff. Explicit implementation of a parallel dictionary. Technical report, Max-Plank Institute für Informatik, September 1996.

A Appendix: ParaDict interfaces

A.1 PP/PL interface, C*

```
void Init (TDict *d);
void Done (TDict *d);
void Clear (TDict *d);
bool Empty (TDict *d);
nat Size (TDict *d);
void Copy (TDict *source, TDict *dest);
TItem:current Insert (TDict *d, TKey:current keys, TInf:current infs);
TItem:current Lookup (TDict *d, TKey:current keys);
void DelItems (TDict *d, TItem:current items);
void Change (TDict *d, TItem:current items, TInf:current infs);
bool:current IsNil (TDict *d, TItem:current items);
TKey:current Keys (TDict *d, TItem:current items);
TInf:current Infs (TDict *d, TItem:current items);
TInf:current Access (TDict *d, TKey:current keys);
void Delete (TDict *d, TKey:current keys);
```

A.2 SP/PL interface, Ansi C

```
void Init_ (TDict *d);
void Done_ (TDict *d);
void Clear_ (TDict *d);
bool Empty_ (TDict *d);
nat Size_ (TDict *d);
void Insert_ (TDict *d, int k, bool mask[], TKey keys[], TInf infs[], TItem *items[]);
void Lookup_ (TDict *d, int k, bool mask[], TKey keys[], TItem items[]);
void DelItems_ (TDict *d, int k, bool mask[], TItem items[]);
void Change_ (TDict *d, int k, bool mask[], TItem items[], TInf infs[]);
void IsNil_ (TDict *d, int k, bool mask[], TItem items[], bool arenil[]);
void Keys_ (TDict *d, int k, bool mask[], TItem items[], TKey keys[]);
void Infs_ (TDict *d, int k, bool mask[], TItem items[], TInf infs[]);
void Access_ (TDict *d, int k, bool mask[], TKey keys[], TInf infs[]);
void Delete_ (TDict *d, int k, bool mask[], TKey keys[]);
```

B Appendix: The rest of the insertion algorithm

The objective of this appendix is twofold: first, it completes the hole insertion algorithm partially presented in Section 5.3; second, it shows how we have refined the algorithms of [18] until a level which can be directly codified in C*.

In section 5.3 we gave the *Insertion* algorithm, which uses the *LeavesUp* procedure to generate hanging request from leaves to the bottom level of the tree and the *HangsUp* procedure that makes climb the hanging requests. Let us now describe in detail these procedures.

Procedure *LeavesUp* creates new hanging requests for the middle elements of each group, rearranges the new splited groups and waits to be called again. This process is depicted in Figure 6. The index used to create new hanging requests is the same of the leaf that is hanged (in this way always different indexes are used and there are no interferences).

```
procedure LeavesUp                                                                (9)
    in leaves :        array [0 .. $k-1$] of $\uparrow$ Node
    in/out nodes :     array [0 .. $k-1$] of $\uparrow$ Node
    in/out heads :     array [0 .. $k-1$] of boolean
    in/out dists :     array [0 .. $k-1$] of natural
    in/out hangs :     array [0 .. $k-1$] of Hang

    var
        m :     array [0 .. $k-1$] of natural
    begin
        for all $i \in [0 .. k-1]$ such that heads[i] do in parallel
            $m[i] := i + dists[i]$ div 2
            — Launch new hangs —
            with hangs[m[i]] do
                act := true; src := leaves[m[i]]; dst := nodes[i] $\uparrow$ .parent; pos := nodes[i] $\uparrow$ .pos
            end with
            heads[m[i]] := false
            if $dists[i] \neq 0$ then        — Become new heads —
                heads[m[i] + 1] := true; nodes[m[i] + 1] := nodes[i]
                $dists[m[i]+1] := dists[i] - dists[i]$ div $2 - 1$
                $dists[i] := dists[i]$ div $2 - 1$; nodes[i] := leaves[m[i]]
            end if
        end for
    end LeavesUp
```

Procedure *HangsUp* recieves an array of hanging requests that have to climb one level of the tree. All of them are processed in parallel, even if they are at different levels of the tree. Due to the design of the algorithm, two consecutive levels of the tree will never have hanging requests.

First of all, this procedure tests if some (unique) request is at the root of the tree. In this case, a new root on top the of the old one must be created. This is the way in which the tree grows in height. After that, all the hanging requests that have to be hanged to the same destination node have to agree to proceed in order. To do that, an ellection is done, thanks to the *urn* field contained in each node and to the fact that hanging requests to the same node must have different positions. Then, according to their positions, hanging requests vote their own identifier in the urns. The last identifier voted will be the secretary and will continue inactive. All other requests will be made inactive, but before that, they transmit their request to their secretary. The last step involves calling *HangUp* to enable that the secretaries hang the sources nodes to the destination ones, creating new nodes if necessary. All this complicated process is made in order not having concurrent reads nor writes and was not given in the original article of [18].

```
procedure HangsUp                                                              (10)
    in/out T :          23Tree
    in/out hangs :      array [0 .. $k-1$] of Hang
    in/out h :          natural

    var
        rqst1, rqst2, rqst3 :   array [0 .. $k-1$] of $\uparrow$ Node
    begin
        — Check root up —
        for all $i \in [0 .. k-1]$ such that $hangs[i].act \wedge hangs[i].dst = $ nil then
            {Number of active processors = 1}
            $RootUp(T, hangs[i].src, T.root)$; $hangs[i].act :=$ false
        end for
        — Communication for exclusive access —
        for all $i \in [0 .. k-1]$ such that $hangs[i].act$ do in parallel
            $rqst1[i] :=$ nil; $rqst2[i] :=$ nil; $rqst3[i] :=$ nil
            — Elect a secretary —
            if $hangs[i].pos = 1$ then $hangs[i].dst \uparrow .urn := i$
            if $hangs[i].pos = 2$ then $hangs[i].dst \uparrow .urn := i$
            if $hangs[i].pos = 3$ then $hangs[i].dst \uparrow .urn := i$
            — Send hanging requests to the secretaries —
            if $hangs[i].pos = 1$ then $rqst1[hangs[i].dst \uparrow .urn] := hangs[i].src$
            if $hangs[i].pos = 2$ then $rqst2[hangs[i].dst \uparrow .urn] := hangs[i].src$
            if $hangs[i].pos = 3$ then $rqst3[hangs[i].dst \uparrow .urn] := hangs[i].src$
            — Elected secretaries hang requests —
            if $\left(\begin{array}{ll} & (hangs[i].pos = 1 \wedge_c hangs[i].dst \uparrow .urn = i) \\ \vee & (hangs[i].pos = 2 \wedge_c hangs[i].dst \uparrow .urn = i) \\ \vee & (hangs[i].pos = 3 \wedge_c hangs[i].dst \uparrow .urn = i) \end{array}\right)$ then
                $HangUp(T, i, hangs[i], rqst1[i], rqst2[i], rqst3[i])$
            else
                $hangs[i].act :=$ false
            end if
        end for
    end HangsUp
```

Finally the last insertion function: *HangUp* is a case analysis to perform the effective hangings. Figure B depicts some cases showing how to resolve them.

```
procedure HangUp                                                               (11)
    in/out T :                  23Tree
    in i :                      natural
    in/out hangs :              array [0 .. $k-1$] of Hang
    in rqst1, rqst2, rqst3 :    $\uparrow$ Node

    var
        nb :     natural
        sons :   $\uparrow$ array [0 .. $6-1$] of Node
        new :    $\uparrow$ Node
    begin
        — Merge sons into sons —
        $nb := 0$
    if $rqst1 \neq$ nil then $sons[nb++] := rqst1$
    $sons[nb++] := hangs[i].dst \uparrow .Lptr$
    if $rqst2 \neq$ nil then $sons[nb++] := rqst2$
    $sons[nb++] := hangs[i].dst \uparrow .Mptr$
    if $rqst3 \neq$ nil then $sons[nb++] := rqst3$
    if $Triple(hangs[i].dst)$ then $sons[nb++] := hangs[i].dst \uparrow .Rptr$
    — Act according with the number of sons —
    if $nb = 3$ then
      $Hang3(hangs[i].dst, sons[0], sons[1], sons[2])$
      $hangs[i].act :=$ false
    else
      $new := NewNode(T)$
      case $nb$ of
        4 $\longrightarrow$ $Hang2(new, sons[0], sons[1])$; $Hang2(hangs[i].dst, sons[2], sons[3])$
        5 $\longrightarrow$ $Hang3(new, sons[0], sons[1], sons[2])$; $Hang2(hangs[i].dst, sons[3], sons[4])$
        6 $\longrightarrow$ $Hang3(new, sons[0], sons[1], sons[2])$; $Hang3(hangs[i].dst, sons[3], sons[4], sons[5])$
      end case
      — Raise hangs[i] request —
      with $hangs[i]$ do
        $pos := hangs[i].dst \uparrow .pos$; $dst := hangs[i].dst \uparrow .parent$; $src := new$
      end with
      — Check root up —
      if $hangs[i].dst \uparrow .parent =$ nil then
        {Number of active processors = 1}
        $RootUp(T, new, T.root)$; $hangs[i].act :=$ false
      end if
    end if
  end HangUp
```

C* implementation

The THang type identifier defined in the following structure corresponds to record 'Hang' of algorithm (6).

⟨*Types for inserting*⟩≡
```
typedef struct {
    bool act;
    TItem src,dst;
    char pos;
} THang;
```

As it can be seen, Insertion corresponds to procedure *Insertion* of algorithm (8):

⟨*Insertion*⟩≡
```
void Insertion (TDict *d, TItem:current nodes, TItem:current leaves) {
    THang:current hangs;
    bool:current heads;
    nat:current dists;

    heads=Ord()==0 || [.-1]nodes!=[.]nodes;
    dists=enumerate(0,CMC_downward,CMC_inclusive,CMC_segment_bit,&heads)-1;
    hangs.act=false;

    LeavesUp (d,leaves,&nodes,&heads,&dists,&hangs);
    while (|= (hangs.act)) {
        HangsUp(d,&hangs);
        HangsUp(d,&hangs);
        LeavesUp (d,leaves,&nodes,&heads,&dists,&hangs);
```

```
      }   }
```

The LeavesUp function implements *LeavesUp* given in algorithm (9).

⟨*Leaves Up*⟩≡

```
      TItem:current LeavesUp ( TDict *d, TItem:current leaves, TItem:current *nodes,
              bool:current *heads, nat:current *dists, THang:current *hangs )
      {
          nat:current m;

          where (*heads) {
              m=Ord() + *dists/2;
              [m]hangs->act=true; [m]hangs->src=[m]leaves;
              [m]hangs->dst=[*nodes]d->cells->parent; [m]hangs->pos=[*nodes]d->cells->pos;
              [m]*heads=false;
              where (*dists!=0) {
                  [m+1]*heads=true; [m+1]*nodes=*nodes;
                  [m+1]*dists=*dists-*dists/2-1;
                  *dists=*dists/2-1; *nodes=[m]leaves;
      }   }   }
```

We give now the implementation of HangsUp and HangsUp2, the functions that make climb the hanging requests, folowing algorithms (10) and (11).

⟨*Hangs Up*⟩≡

```
      void HangsUp (TDict *d, THang:current *hangs) {
          nat:SCells urn;
          TItem:current rqst1,rqst2,rqst3;

          where (hangs->act && hangs->dst==NIL) {
              if (SomeActive()) {
                  RootUp(d,(TItem)hangs->src,d->root);
                  hangs->act=false;
          }   }
          where (hangs->act) {
              rqst1=NIL; rqst2=NIL; rqst3=NIL;

              where (hangs->pos==1) [hangs->dst]urn=Ord();
              else where (hangs->pos==2) [hangs->dst]urn=Ord();
              else where (hangs->pos==3) [hangs->dst]urn=Ord();

              where (hangs->pos==1) [[hangs->dst]urn]rqst1=hangs->src;
              else where (hangs->pos==2) [[hangs->dst]urn]rqst2=hangs->src;
              else where (hangs->pos==3) [[hangs->dst]urn]rqst3=hangs->src;

              where (      (hangs->pos==1 && [hangs->dst]urn==Ord())
                       ||  (hangs->pos==2 && [hangs->dst]urn==Ord())
                       ||  (hangs->pos==3 && [hangs->dst]urn==Ord()))
                  HangsUp2(d,hangs,rqst1,rqst2,rqst3);
              else
                  hangs->act=false;
      }   }
```

It can be remarked that in the C* implementation the urn variables are declared as local variables instead of being stored in the nodes. We took that decision in order to save memory space.

⟨*Hangs Up2*⟩≡

```
      void HangsUp2 ( TDict *d, THang:current *hangs,
        TItem:current rqst1, TItem:current rqst2,TItem:current rqst3 )
{
    char:current nbs;         TItem:current sons[6];
    TItem:current news;       TCell:current cells;

    cells=[hangs->dst]*d->cells; nbs=0;
    where (rqst1!=NIL) sons[nbs++]=rqst1;
    sons[nbs++]=cells.Lp;
    where (rqst2!=NIL) sons[nbs++]=rqst2;
    sons[nbs++]=cells.Mp;
    where (rqst3!=NIL) sons[nbs++]=rqst3;
    where (cells.sons==3) sons[nbs++]=cells.Rp;

    where (nbs==3) {
        Hang3s(d,hangs->dst,sons[0],sons[1],sons[2]);
        hangs->act=false;
    } else {
        news=AllocateCells(d);
        where (nbs==4) {
            Hang2s(d,news,sons[0],sons[1]);
            Hang2s(d,hangs->dst,sons[2],sons[3]);
        } else where (nbs==5) {
            Hang3s(d,news,sons[0],sons[1],sons[2]);
            Hang2s(d,hangs->dst,sons[3],sons[4]);
        } else where (nbs==6) {
            Hang3s(d,news,sons[0],sons[1],sons[2]);
            Hang3s(d,hangs->dst,sons[3],sons[4],sons[5]);
        }
        hangs->pos=[hangs->dst]d->cells->pos;
        hangs->dst=[hangs->dst]d->cells->parent;
        hangs->src=news;
        where (hangs->dst==NIL) if (SomeActive()) {
            RootUp(d,(TItem)hangs->src,d->root);
            hangs->act=false;
}   }   }
```

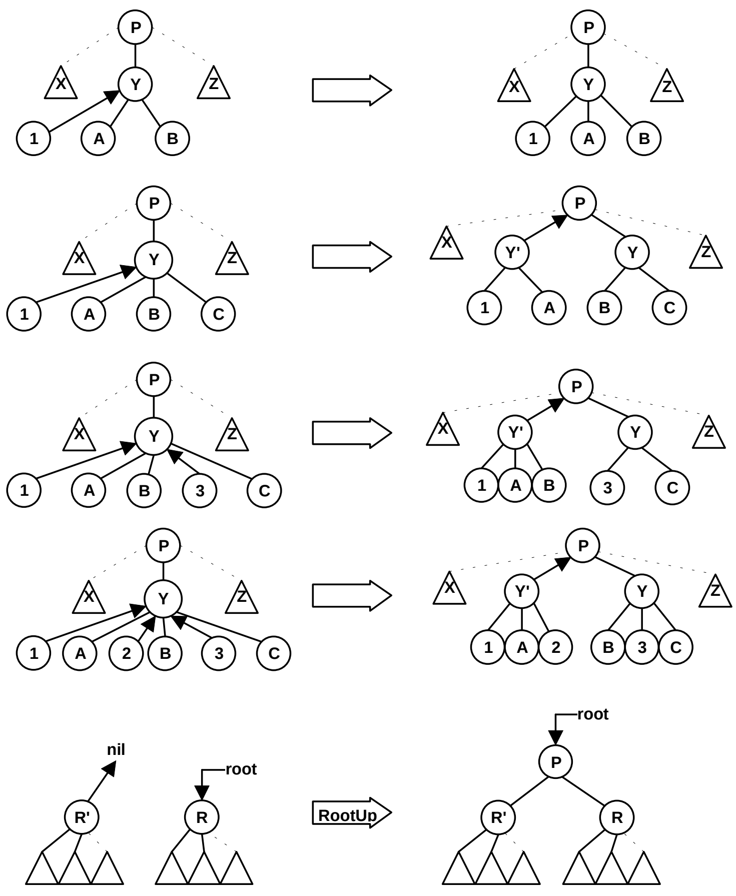

Figure 11: Processing hanging requests in the insertion: selected cases.