A Sequential and Parallel Implementation of Skip Lists

Xavier Messeguer

November 7, 1994

1 Introduction

This Research Report deals with the sequential and the parallel inplementation of Skip Lists and its relative performance. The sequential algorithm is codified in `C` and the parallel one in the parallel extension `C*`.

The sequential implementation is taken as a first step for the parallel one. For instance, although the sequential case suggest to implement the skip list with pointers, we have choose arrays because they are close to the parallel data implementation. The sequential program was executed in a SUN Sparc Station. The parallel one in a CM-200 with 2024 processors and a SUN Sparc Station as a Front End.

We analyse the performance of the insertion algorithms in order to determine the running time. The sequential algorithm running time is given by $t = S_1 n \log n + S_2 n + S_3$ and n is the number of random keys to be inserted into an empty skip list. In the parallel algorithm the running time is $t = P_1 \log n + P_2 \log k + P_3$ where k is either the number of keys to be inserted into an skip list of size n and the number of processors allocated.

2 Sequential algorithm

2.1 Implementation

The code of the program has been divided in the following parts:

$\langle * \rangle \equiv$

$\langle$*includes*$\rangle$
$\langle$*defines*$\rangle$
$\langle$*general types*$\rangle$
$\langle$*skip list types*$\rangle$
$\langle$*packet type*$\rangle$
$\langle$*specification of procedures*$\rangle$
$\langle$*main program*$\rangle$
$\langle$*implementation of search procedures*$\rangle$
$\langle$*implementation of insert procedures*$\rangle$
$\langle$*implementation of delete procedures*$\rangle$
$\langle$*implementation of general procedures*$\rangle$

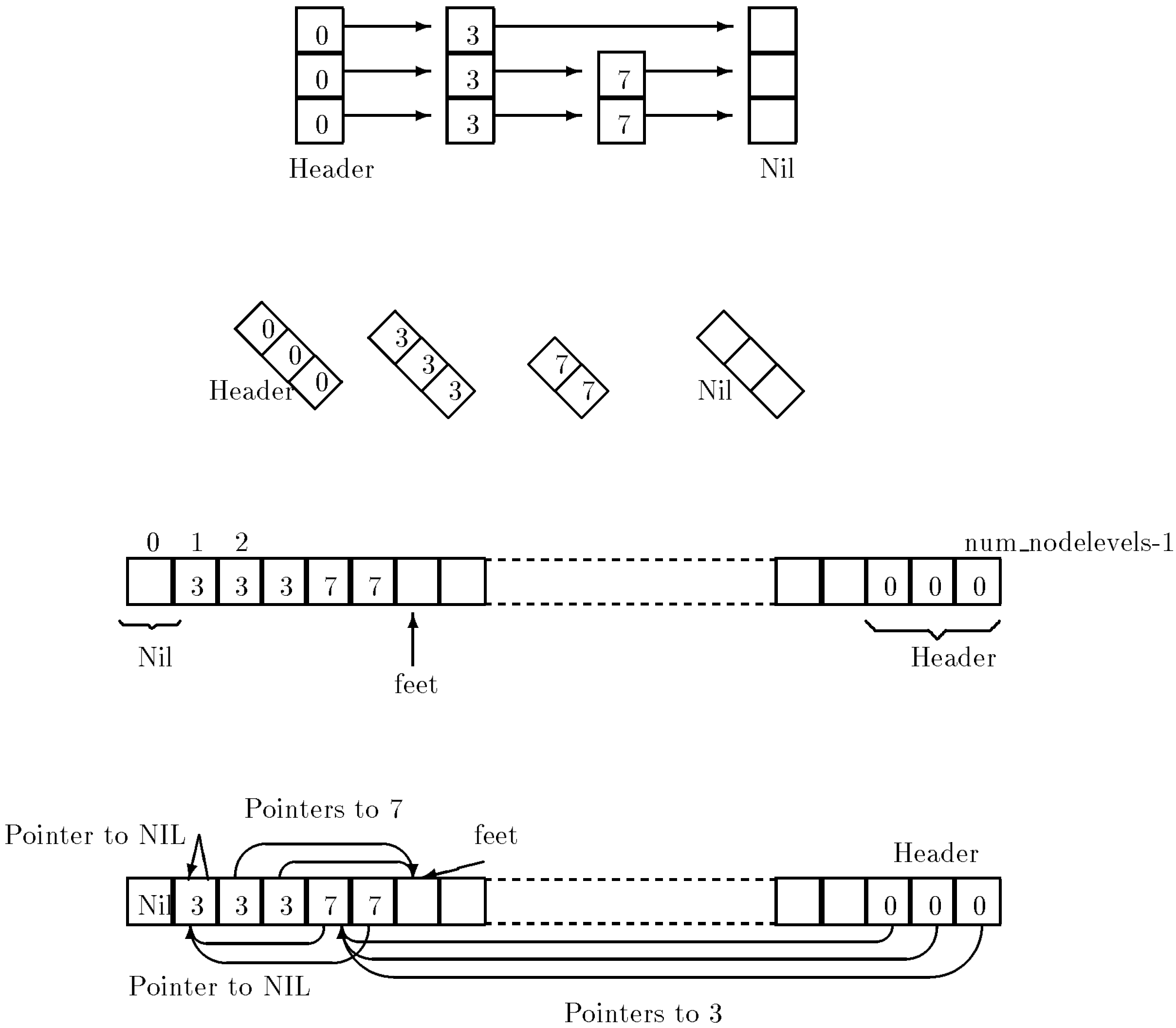

Figure 1: The meaning of the `skip list` array.

The explanations have been developed in a different order. Firstly we explain the data types and the implementation of procedures, and the remainder parts, needed for the execution of the program, are explained later.

2.1.1 Skip list types

The skip list was implemented in an array of `nodelevel_elem` data type. This type is composed by two fields: the `key` field that stores the value of items and the `forward` field that points to a `nodelevel_elem` (the forward node).

⟨*skip list types*⟩≡
```
typedef int ref_node;
typedef int key_type;
typedef struct {
            key_type key;
```

```
                ref_node forward;
               } nodelevel_elem;
typedef nodelevel_elem skiplist_array[ num_nodelevels];
⟨skip list type⟩
⟨update type⟩
```

The meaning of the skiplist_array is shown in figure 1. The nodes have been layed down horizontally and located consecutivelly except the header and the NIL node: the header was located at the end of the array and the NIL at the begining. This implementation is determined by the following facts:

- The array could be easily considered as an array of processors in the parallel implementation in the CM.
- The height of the header can change during the inserting process, then the node/levels should be easily added or kept.

The skiplist_type needs some escalar magnitudes:

- header: stores a pointer to the header.
- feet: stores a pointer to the first empty place of the array.
- number: stores the number of stored items in the skip list.
- level: stores the level of the skip list.

Then

⟨skip list type⟩≡
```
typedef struct {
                skiplist_array skiplist;
                ref_node header,feet;
                int level,number;
               } skiplist_type;
```

Finally we implement the array of updates:

⟨update type⟩≡
```
typedef ref_node update_array[M];
```

2.1.2 Packet type

Although in this sequential implementation the type `packet` does not has an important rol, lets us consider it as a simplification of the parallel type.

When a key is going to be searched, the `key` field stores the value, the `node` and `level` points to the current node/level of the skip list and the field `active` checks the search process.

⟨*packet type*⟩≡
```
typedef struct {
      key_type key;
      ref_node node;
      int level;
      boolean active;
     } packet;
```

2.1.3 General types

We define the boolean types only.

⟨*general types*⟩≡
```
typedef int boolean;
#define false 0
#define true 1
```

2.1.4 Search procedures

The procedure search routes a packet into the skip list until the smaller item of the skip list, greater or equal than `a`, is found. The procedure returns a pointer to this item (a pointer to a node).

⟨*implementation of search procedures*⟩≡
```
void search(key_type a,skiplist_type *S,ref_node *node)
{
  packet p;

  init_search(a,S,&p);
  while (p.active) route(&p,S);
  *node=p.node;
}
```
⟨*init_search*⟩
⟨*route*⟩

The procedure `init_search` determines the initial value of the packet: it is located at `S->header` with level `S->level` and is active.

⟨*init_search*⟩≡

```
void init_search(key_type a ,skiplist_type *S ,packet *p )
{
      p->key=a;
      p->node=S->header;
      p->level=S->level;
      p->active=(p->level>0);
}
```

The procedure `route` moves the packet. It compares its key `a` against the forward key of the current item `b` and sends the packetd down or forward.

⟨*route*⟩≡

```
void route ( packet *p,skiplist_type *S )
{
   key_type b;
   ref_node f;

   f=forward(S,p->node,p->level);
   b=key(S,f);
   if (b<p->key)  push_forward(p,f);
     else push_down(p);
}
```

The procedure `forward` returns a pointer to the forward node `p->node` at level `p->level`. The procedure `key` returns the value stored at node pointed to by `f`, and the `push_forward` and `push_down` locates the packet at the new node/level. All this procedures are explained in following sections.

2.1.5 Insert procedures

This procedure `insert` inserts a key `a` in the skip list `S`.

⟨*implementation of insert procedures*⟩≡

```
void insert(key_type a,skiplist_type *S)
{
      update_array update;
      ref_node node,newnode;
      int levelnewnode;

      search_with_update(a,S,&node,update);
      if (key(S,forward(S,node,1))!=a) {
              create_new_node(S,&newnode,&levelnewnode,update);
              merge(S,a,newnode,levelnewnode,update);
       }
}
```

⟨*search_with_update*⟩
⟨*create_new_node*⟩
⟨*merge*⟩

The procedure `search_with_update` returns either a pointer to a node and an `update` array. This array is updated by the procedure `route_with_update`.

⟨*search_with_update*⟩≡

```
void search_with_update(key_type a, skiplist_type *S, ref_node *node ,
                      update_array update)
{
  packet p;

  init_search(a,S,&p);
  while (p.active) route_with_update(&p,S,update);
  *node=p.node;
}
```

⟨*route_with_update*⟩

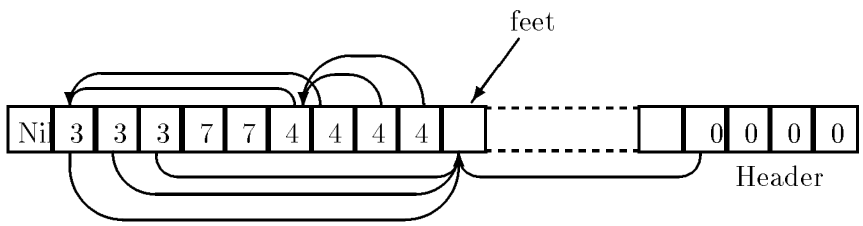

Figure 2: Inserting the key 4

⟨*route_with_update*⟩≡

```
void route_with_update ( packet *p,skiplist_type *S,
                         update_array update)
{
  key_type b;
  ref_node f;

  f=forward(S,p->node,p->level);
  b=key(S,f);
      if (b<p->key)  push_forward(p,f);
        else {
              update[p->level]=p->node;
              push_down(p);
        }
}
```

The figure 2 shows how the array of figure 1 should be modified by the insertion of key 4.

The `create_new_node` procedure assigns radomly a level to the new node and returns a pointer to `newnode`. If the level of the new node is greather than the level of the skip list then the `header` grows up and the `update` is adjusted.

⟨*create_new_node*⟩≡

```
void create_new_node(skiplist_type *S,ref_node *newnode , int *level,
                 update_array update)
{
      int i;

      *level=random_level(.5);
  /* The feet is traslated to right */
      S->feet=S->feet+*level;

  /* The pointer to the new node and feet have the same value */
      *newnode=S->feet;

      if (S->level<*level){
        /* The new positions of the header should be initialized */
              for(i=S->header-S->level-1;i>=S->header-(*level);i--){
                                S->skiplist[i].forward=NIL;
                                S->skiplist[i].key=minkey;
                       }
        /* The updates must be initialized */
              for (i=*level;i>S->level;i--) update[i]=S->header;
        /* The skip list also grows */
              S->level=*level;
      }
  /* The size has increased */
      S->number++;
}
```

The merge procedure updates the value a and determines the new pointers.

⟨*merge*⟩≡

```
void merge(skiplist_type *S,key_type a, ref_node newnode,
         int levelnewnode, update_array update)
{
      int l;

      for (l=1;l<=levelnewnode;l++){
              S->skiplist[newnode-l].forward=
                             forward(S,update[l],l);
   /* The node/levels pointed by the new node */
              S->skiplist[newnode-l].key=a;
   /* The value of the new node */
              S->skiplist[update[l]-l].forward=newnode;
   /* The node/levels that points to the new one inserted */
              }
}
```

2.1.6 Delete procedures.

When a key a is going to be deleted, firstly it should be searched. If it is found then it is removed from the skip list and the memory position is deallocated.

⟨*implementation of delete procedures*⟩≡

```
void delete(key_type a,skiplist_type *S)
{
      update_array update;
      ref_node f,node;
      int nodelevelsfree;

      search_with_update(a,S,&node,update);
      f=forward(S,node,1);
      if (key(S,f)==a) {
          remove(S,f,update, &nodelevelsfree);
          deallocate_memory(S,f,nodelevelsfree);
      }
}
```

⟨*remove*⟩
⟨*deallocate_memory*⟩

The procedure `remove` restructures the level and the pointers of the skip list leaving a sequence of empty places in the array.

⟨*remove*⟩≡

```
void remove(skiplist_type *S,ref_node node, update_array update,
                int *nodelevelfree)
{
      int i,level;

      level=level_of_node(S,node);
      for (i=1;i<=level;i++)
          S->skiplist[update[i]-i].forward=
                           forward(S,forward(S,update[i],i),i);
  /* The node/levels that points to the removed node, have been
     pointed to the forward node/levels of the removed one /*
        while (S->level>1 & forward(S,S->header,S->level)==NIL)
         (S->level)--;
  /* If some levels of the header points to NIL the level of the
     skip list must decrease */
        S->number--;
  /* The number of elements of the skiplist has decreased */
        *nodelevelfree=level;
  /* A pointer to the empty sequence of places */
 }


int level_of_node( skiplist_type *S, ref_node node)
{
      int level;
      key_type a;

      a=key(S,node);
      level=1;
      while (S->skiplist[node-level].key==a) level++;
      return level-1;
}
```

Finally the procedure `deallocate_memory` that deallocates the removed node is implemented as a traslation of the elements in the skip array. In order to understand the procedure we divide the array in five parts:

- Left part: the places of the array located at the left of the removed node.
- Removed part: the places of the array of the removed node.
- Right part: the places of the array located at the right of the removed node until the feet.
- Empty part: the places of the array located between the feet and the header.
- Header part: the places of the array of the header.

⟨*deallocate_memory*⟩≡

```
void deallocate_memory( skiplist_type *S,ref_node node, int nodelevelfree)
{
      int i;


      for (i=node;i<S->feet;i++)
         S->skiplist[i-nodelevelfree]=S->skiplist[i];
  /* The right part is traslated to the left nodelevelfree unities */
      S->feet=S->feet-nodelevelfree;
  /* The new feet is determined */
      for (i=1;i<S->feet;i++)
        if (S->skiplist[i].forward>node-nodelevelfree)
           S->skiplist[i].forward=S->skiplist[i].forward-nodelevelfree;
  /* The pointers to the right part are recalculated */
      for (i=S->header-S->level;i<= num_nodelevels;i++)
        if (S->skiplist[i].forward>=node)
           S->skiplist[i].forward=S->skiplist[i].forward-nodelevelfree;
  /* The pointers of the header also are recalculated */
}
```

2.1.7 General procedures

We explain in this section the low level procedures of the skip list and some other mathematical ones.

⟨*implementation of general procedures*⟩≡

⟨*empty_skiplist*⟩
⟨*forward*⟩
⟨*key*⟩
⟨*push_forward*⟩
⟨*push_down*⟩
⟨*random_level*⟩
⟨*log*⟩

The procedure `empty_skiplist` initializes an empty skip list letting level 1 and only two nodes: the header with the minimum key and the NIL with the maximum key.

⟨*empty_skiplist*⟩≡
```
void empty_skiplist (skiplist_type *S)
{
       S->header= num_nodelevels;
  /* The length of the array is  num_nodelevels */
       S->feet=1;
       S->level=1;
       S->number=0;
       S->skiplist[S->header-1].forward=NIL;
       S->skiplist[S->header-1].key=minkey;
       S->skiplist[NIL-1].key=maxkey;
}
```

The `forward` procedure return a pointer to the node `f` at level `l`.

⟨*forward*⟩≡
```
ref_node forward(skiplist_type *S,ref_node f, int l)
{
   return S->skiplist[f-l].forward;
}
```

The `key` returns the keyy of the node `f`.

⟨*key*⟩≡
```
key_type key(skiplist_type *S,ref_node f)
{
   return S->skiplist[f-1].key;
}
```

The `push_forward` and the `push_down` moves the packets forward or down the skip list.

⟨*push_forward*⟩≡
```
void push_forward(packet *p, ref_node f)
{
  p->node=f;
}
```

⟨*push_down*⟩≡
```
void push_down(packet *p)
{
  if (p->level==1) p->active=false;
    else p->level--;
}
```

Finally, the following procedures calculates the random level with probabilty `p` and the logaritm of `n`.

⟨*random_level*⟩≡

```
int random_level(float p)
{
   int level;

   level=1;
   while (rand()%10>=(int)(p*10)) level++;
   return level;
}
```

⟨*log*⟩≡

```
int log(int n)
{
   int l;

   l=0;
   while (n>1) {l++;n=n/2;}
  return l;
}
```

2.1.8 Specification of procedures

The specification of the procedures is:

⟨*specification of procedures*⟩≡
  ⟨*specification of search procedures*⟩
  ⟨*specification of insert procedures*⟩
  ⟨*specification of delete procedures*⟩
  ⟨*specification of general procedures*⟩

where

⟨*specification of search procedures*⟩≡

```
void search(key_type ,skiplist_type *, ref_node *);
void init_search(key_type,skiplist_type * ,packet *);
void route ( packet *,skiplist_type *);
```

⟨*specification of insert procedures*⟩≡

```
void insert(key_type,skiplist_type * );
void search_with_update(key_type ,skiplist_type *, ref_node * ,
                        update_array );
void create_new_node(skiplist_type *,ref_node *,int *,update_array);
void merge(skiplist_type *, key_type , ref_node, int, update_array);
void route_with_update ( packet *,skiplist_type *, update_array);
```

⟨*specification of delete procedures*⟩≡

```
void delete(key_type,skiplist_type * );
void deallocate_memory( skiplist_type *,ref_node, int);
void remove( skiplist_type *,ref_node, update_array , int *);
```

⟨*specification of general procedures*⟩≡

```
void empty_skiplist (  skiplist_type * );
void push_down(packet *);
void push_forward(packet *, ref_node);
ref_node forward( skiplist_type *,ref_node,int) ;
key_type key(skiplist_type *, ref_node f);
int level_of_node(skiplist_type *, ref_node);
int random_level(float );
int log(int);
```

2.1.9 Remainder code

The `main` program inserts `number_of_insertions` keys radomly computed into an empty skip list.

⟨*main program*⟩≡

```
main ()
{

   skiplist_type S;
   key_type a;
   int i;

   empty_skiplist(&S);
   for (i=1;i<=number_of_insertions;i++) insert(rand()%maxkey,&S);
}
```

In order to execute the program some external libraries are needed

⟨*includes*⟩≡

```
#include <stdio.h>
#include <stdlib.h>
```

and some constant should be initialized

⟨*defines*⟩≡

```
  #define num_nodelevels 1000000
/* The length of the skip array */
  #define M 20
/*The height of the update array */
  #define NIL 1
  #define minkey 0
  #define maxkey num_nodelevels
  #define number_of_insertions 1000
```

Name	*% time*	*Self*	*Descen.*	*Called*	*Parent*
main	89.7	0.02	2.67	1	
insert	87.7	0.06	2.57	10000	main
search	74.1	0.24	1.98	10000	insert
route	65.1	0.64	1.31	227586	search
forward	16.5	0.50	0.00	10000+19333+227586	insert,merge,route
key	16.3	0.49	0.00	10000+227586	insert,route
push_down	9.8	0.30	0.00	128448	route
merge	5.9	0.14	0.04	9953	insert
create	4.4	0.06	0.07	9953	insert
push_forward	3.7	0.11	0.00	99138	route
random_level	2.4	0.00	0.07	9953	main
init_search	1.0	0.03	0.00	10000	search

Table 1: Profile information

2.2 Performace of sequential algorithm

2.2.1 A first approximation to data

The data was obtained with a programming utility that produces a profile execution of the programm. For each routine the following data are given:

- **%time**: the percentage of the total time of the program accounted for by this function and its descendents.
- **Self**: the number of seconds spent in this function itself.
- **Descendents**: the number of seconds spent in the descendents of this function on behalf of this function.
- **Called**: the number of times this function is called (other than recursive calls).
- **Parents**: the names of the parents.

The table 1 concerns to an insertion of 10000 items into an empty skip list, the rows are sorted by the summ of *self* and *descendents*. For instance, the routine *search_with_update* performs 74.1% of the total running time, spends 0.24 seconds on itself and 1.98 seconds on its descendents, and is called 10.000 times by its parent routine *insert*.

2.2.2 The accuracy of data

The time sharing forces us to work with samples, whose size should be determined according to the desired relative error of values. For instance, the table 2 contains the time spent (in seconds) by the function *insert* when 32000 items have been inserted 100 times. This sample has the statistical magnitudes:

$$\text{Average}=9.7043 \qquad \text{Variance}=0.0435845$$

9.23	9.55	9.58	9.39	9.38	9.73	9.73	9.42	9.82	9.56
9.4	19.43	9.46	9.51	10.01	10.23	9.51	9.6	9.75	10.07
9.89	9.71	9.82	9.32	9.9	9.89	9.39	10.01	9.76	9.68
9.88	9.94	9.97	9.69	9.63	9.95	9.77	9.36	9.73	9.76
9.7	9.34	9.48	9.55	9.63	9.71	9.98	9.86	9.78	9.95
9.7	9.86	9.68	9.81	9.77	9.66	9.83	9.51	9.78	9.43
10.04	9.77	9.48	9.8	9.7	9.46	9.89	9.45	10	10.14
9.74	9.69	9.77	9.8	9.91	9.42	9.81	9.71	9.68	9.7
9.4	9.45	9.68	9.52	10	9.73	9.62	9.77	9.74	9.71
9.55	10.22	9.59	9.75	9.46	9.83	9.64	9.5	9.97	9.82

Table 2: Time spent in the insercions routine

frequency	1	2	12	13	8	23	17	12	8	2	2
value	9.2	9.3	9.4	9.5	9.6	9.7	9.8	9.9	10	10.1	10.2

Table 3: Frequency function

The table 3 contains the frequency of data grouped by small intervals and suggest that they are close to a normal distribution. Therefore, the following formula

$$\text{Prob}\left(\mu \in \left[9.7043 - 1.96 \cdot \sqrt{\frac{0.0435845}{n}}, 9.7043 + 1.96 \cdot \sqrt{\frac{0.0.0435845}{n}}\right]\right) \geq 95\%$$

can be applied to obtain a good aproximation between the size of the sample and the accuracy of the average. For size 25 the probability that the average value of a sample belongs to the interval $[9.7043 - 0.0818, 9.7043 + 0.0818]$ is sufficiently hight (95%). Then we conclude that a sample of size 25 sufices to determine the performance score of the all functions defined in the sequential algorithm.

2.2.3 The data

We have executed the insertion procedure for several sets of input data, and the time spent by the procedure *insert* has took into account. For instance, the first column of the table 4 shows the sum of *self* and *descendents* running times, and the figure 3 plots the data in a logaritmic scale.

2.2.4 Analysis of data

The time spend to insert n items into an empty skip list can be aproximated by

$$t_n = S_1 \cdot n \log n + S_2 n + S_3$$

The constants S_1 is given by

$$S_1 = \lim_{n \to \infty} \frac{t_n}{n \cdot \log n}$$

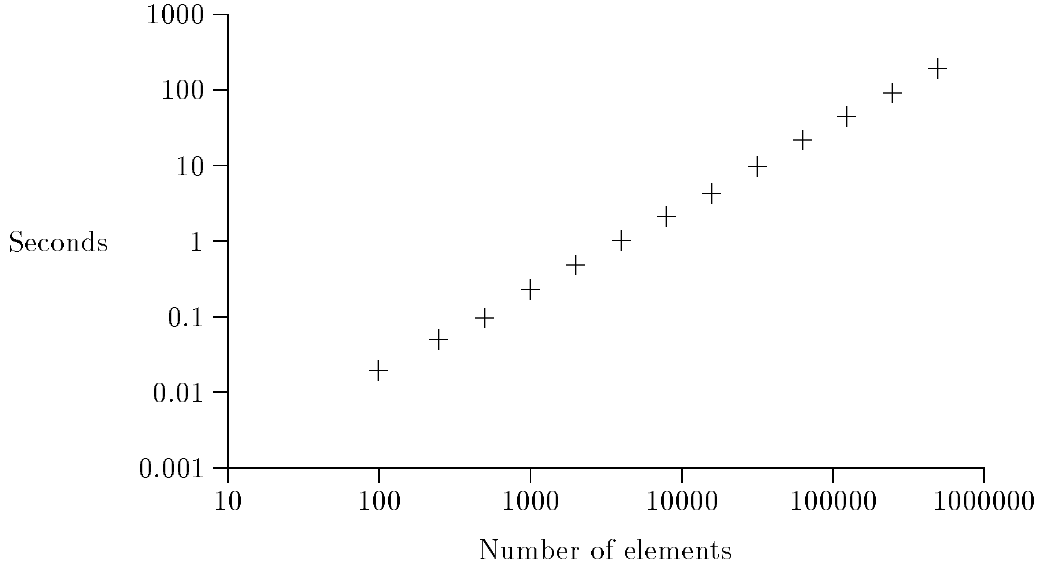

Figure 3: Sequential insertions in an empty Skip List

The second column of the table 4 shows the sequence of values: let $S_1 = 2.073 \cdot 10^{-5}$. The third column, that has the relative error induced by S_1 over t_n, clearly shows that it is smaller than our approximation (less than 1%). Then we conclude with

$$t_n = 2.07310^{-5} n \log n$$

.

3 Parallel algorithm

3.1 Implementation

The code of the program has been divided in the following parts:

⟨*⟩≡
  ⟨*includes*⟩
  ⟨*defines*⟩
  ⟨*sequential types*⟩
  ⟨*CM skip list types*⟩
  ⟨*FE skip list types*⟩
  ⟨*CM packet type*⟩
  ⟨*CM update array*⟩
  ⟨*specification of CM procedures*⟩
  ⟨*main program*⟩
  ⟨*implementation of CM search procedures*⟩
  ⟨*implementation of CM insert procedures*⟩
  ⟨*implementation of CM general procedures*⟩

k	t_n	C_1	$error$
125	0.0196	2.2510e-05	1.23993e-05
250	0.0508	2.5509e-05	3.80693e-05
500	0.976	2.1772e-05	9.33929e-06
1000	0.234	2.3480e-05	2.74093e-05
2000	0.486	2.2160e-05	1.56793e-05
4000	1.02	2.1269e-05	6.44929e-06
8000	2.12	2.04577e-05	-3.53071e-06
16000	4.36	1.9516e-05	-1.69482e-05
32000	9.74	2.0538e-05	-2.86571e-06
64000	21.7	2.1294e-05	8.99804e-06
125000	44.5	2.1029e-05	5.05658e-06
250000	93.0	2.0740e-05	1.82584e-07
500000	196	2.0735e-05	8.65836e-08
		0.00002073	

Table 4: Time spent to inser k items

3.1.1 Sequential types

The `nodelevel_elem` type of the sequential case is renamed as `nodelevel_CM` and the boolean remains unchanged.

⟨*sequential types*⟩≡
```
typedef int ref_node;
typedef int key_type;
typedef struct {
              key_type key;
              ref_node forward;
             } nodelevel_CM;

typedef int boolean;
```

3.1.2 Skip list types

The array type `skiplist_array` of the sequential case is also applied in the parallel case but allocating a processor at each cell of the array. That structure defines a *shape* in `C*`. Thus `skiplist_CM` type is a shape whose cells belongs to `nodelevel_CM` type.

⟨*CM skip list types*⟩≡
```
shape [number_nodelevels] skiplist_shape;
typedef nodelevel_CM:skiplist_shape skiplist_CM;
```

The skip list data type needs some scalar magnitudes that take place in the front end.

⟨*FE skip list types*⟩≡
```
typedef struct {
              ref_node header,feet;
              int level,number;
             } skiplist_FE;
```

The meaning of this magnitudes is the same as the sequential case (See fig. 1 of section 2.1)

3.1.3 Packet type

The `set_of_packets` has been implemented with another shape denoted `packet_shape`. Its cells belongs to `packet_CM` type. This last type contains the well known sequentials fields `key`, `node`, `level`, `active` and two new ones `first` and `last` that points to first and to last keys of the packet.

⟨*CM packet type*⟩≡
```
shape [MAXk] packet_shape;
typedef  int ref_key;
typedef struct {
      key_type key;
      ref_node node;
      int level;
      boolean active;
      ref_key first, last;
     } packet_CM;
typedef packet_CM:packet_shape set_of_packets;
```

3.1.4 Update array

This squared array is implemented with a `packet_shape` of pointers with height $M + 2$.

⟨*CM update array*⟩≡
```
typedef ref_node:packet_shape  M_array_CM[M+2];
```

3.1.5 CM search procedures

The procedure search_CM searchs the keys a, that contains $a_1, \cdots, a_k$, in the skip list and returns a pointer to the nodes (in node).

⟨*implementation of CM search procedures*⟩≡

```
void search_CM( key_type:packet_shape *a,
       ref_key k, s            kiplist_CM *S,
       skiplist_FE Sfe,        ref_node:packet_shape *node )
{

     set_of_packets P;
     int i,runsize;

     runsize=log(k);

     init_search_CM(*a,k,Sfe,&P);
     with (packet_shape)
        while (|=(P.active))
           for(i=0;i<runsize;i++)
              where (P.active)
                 route_CM(&P,S);
 /* The first key of each packet has the searched pointer */
     *node=spread_ref_node_CM(P.node);
 /* The pointer has been spread to all keys of the packet */
}
```

⟨*init_search_CM*⟩
⟨*route_CM*⟩

The procedure init_search_CM determines the initial values of the set_of_packets.

⟨*init_search_CM*⟩≡

```
 void init_search_CM( key_type:packet_shape a,         ref_key k,
       skiplist_FE Sfe,        set_of_packets *P  )
 {
     with (packet_shape) {
       ([1]*P).first=1;
       ([1]*P).last=k;
   /* There are only one packet with length k */
       P->active=false;
       ([1]*P).active=true;
       ([1]*P).level=Sfe.level;
       P->node=0;
       ([1]*P).node=Sfe.header;
       P->key=a;
      }
 }
```

The procedure route_CM does not contains only one alternative sentence as the sequential case, it has a sequential composition of three alternatives represented by the statement where.

⟨*route_CM*⟩≡

```
void route_CM( set_of_packets *P,
               skiplist_CM *S     )
{
   /* packet_shape is the current shape */
   /* P->active is the current context */

     key_type:packet_shape b;
     ref_node:packet_shape f;

     f=[P->node-P->level] (S->forward);
   /* f contains the forward node of each active packet */
     b=[f-1](S->key);
   /* b contains the key of the forward node*/
     where (b<[P->first]P->key) push_forward_CM(P,f);
     where (b>=[P->last]P->key) push_down_CM(P);
     where (b>=[P->first]P->key && b<[P->last]P->key) split_CM(P);
}
```

3.1.6 CM insert procedures

The following procedure inserts the array a with length k in the skip list.

⟨*implementation of CM insert procedures*⟩≡

```
void insert_CM(key_type:packet_shape *a,      ref_key k,
               skiplist_CM *S,                skiplist_FE *Sfe)
{
      ref_node:packet_shape node,newnode;
      M_array_CM update,succ;
      int:packet_shape level;
      int newmaxlevel;

      search_with_update_CM(a,k,S,Sfe,&node,update);
      create_new_node_CM(a,S,Sfe,&newnode,&level,&newmaxlevel,k);
      make_succ_CM(&level,succ,k);
      merge_CM(S,Sfe,/*&node,*/&newnode,&level,update,succ,&newmaxlevel,k);
}
```

⟨*search_with_update_CM*⟩
⟨*create_new_node_CM*⟩
⟨*make_succ_CM*⟩
⟨*merge_CM*⟩

The procedure `search_with_update_CM` returns the set of pointers `node` and the `update` array.

⟨*search_with_update_CM*⟩≡

```
void search_with_update_CM(key_type:packet_shape *a,
      ref_key k,              skiplist_CM *S,
      skiplist_FE *Sfe,       ref_node:packet_shape *node,
      M_array_CM update )
{
     set_of_packets P;
     int l,i,runsize;

     runsize=log(k);

     init_search_CM(*a,k,*Sfe,&P);
     for(l=1;l<=M;l++) update[l]=0;
       with (packet_shape)
          while (|=(P.active))
             for(i=0;i<runsize;i++)
                where (P.active)
                  route_with_update_CM(&P,S,update);
     *node=spread_ref_node_CM(P.node);
     for(l=1;l<=M;l++) update[l]=spread_ref_node_CM(update[l]);
}
```

⟨*route_with_update_CM*⟩

⟨*route_with_update_CM*⟩≡

```
void route_with_update_CM ( set_of_packets *P,
      skiplist_CM *S,       M_array_CM update)
{
  /* packet_shape is the current shape */
  /* P->active is the current context */

    key_type:packet_shape b,b1,b2;
    ref_node:packet_shape f;

    f=[P->node-P->level] (S->forward);
    b=[f-1](S->key);
    b1=[P->first]P->key;
    b2=[P->last]P->key;

    where (b<b1) push_forward_CM(P,f);
    where (b>=b2) {update[P->level]=P->node;push_down_CM(P);}
    where (b>=b1 && b<b2) if (|=(P->active)) split_CM(P);
}
```

The first part of the procedure create_new_node_CM determines the cells of the array shape that will be assigned to new nodes. The second part updates the values and the last part restructures the skip list if its level has grow.

⟨*create_new_node_CM*⟩≡

```
void create_new_node_CM(key_type:packet_shape *a,
          skiplist_CM *S,              skiplist_FE *Sfe,
          ref_node:packet_shape *newnode,
    /* newnode points to (newnode,*level) node/level */
          int:packet_shape *level, int *newmaxlevel,
          int k)
    /* Current shape is packet_shape */
{
      int i;

      *level=random_level_CM(.5);
   /* Only the cells form 0 to k can be a level greather than zero */
      where (pcoord(0)==0 || pcoord(0)>k)*level=0;
   /* This recuction computes the maximum level */
      *newmaxlevel=(>?=*level);
   /* The procedure scan makes a prefix sum of levels */
      *newnode=scan(*level,0,CMC_combiner_add,CMC_upward,
                    CMC_none,CMC_no_field,CMC_inclusive);
      *newnode=*newnode+Sfe->feet;
   /* newnode contains the cell of the shape where the nodes
      will be inserted */

      for (i=1 ;i<=*newmaxlevel;i++)
           where (*level>=i)
              [*newnode-i]S->key=*a;
   /* The keys have been updated in all node/levels */
   /* The feet and the number are actualized */
      Sfe->feet=Sfe->feet+(+=*level);
      Sfe->number=Sfe->number+k;

   /* If the level of the newnode is greather than the level of S */
      if (Sfe->level<*newmaxlevel)
        with (skiplist_shape)
          where ((pcoord(0)>=Sfe->header-*newmaxlevel) &&
                      (pcoord(0)<Sfe->header-Sfe->level)){
              S->forward=NIL;
              S->key=minkey;
          }
       /* The new positions(levels) of the header have been determined */
}
```

The procedure make_succ_CM builds a skip list in the array succ.

⟨*make_succ_CM*⟩≡
```
void make_succ_CM( int:packet_shape *level,
      M_array_CM succ,                int k )
{
   ref_node:packet_shape succ_row;
   bool:packet_shape sbit;
   int l;

   [0]*level=M;
   [k+1]*level=M+1;
   for (l=1;l<=M;l++) {
          sbit=0;succ_row=1;
          where (*level>=l) {sbit=1;succ_row=0;}
    /* sbit is a mask and succ_row the ith row of succ */
          where (pcoord(0)<=k+1)
            succ_row=scan(succ_row,0,CMC_combiner_add,CMC_downward,
                  CMC_start_bit,&sbit,CMC_inclusive);
    /* scans makes a prefix sum over the segment defined with a mask */
          succ[l]=0;
          succ[l]=[.+1]succ_row+pcoord(0)+1;
          where (sbit==0) succ[l]=0;
   }
}
```

The following procedure merges the skip list S with the succ skip list.

⟨*merge_CM*⟩≡
```
void merge_CM (skiplist_CM *S,        skiplist_FE *Sfe,
      ref_node:packet_shape *newnode, int:packet_shape *level,
      M_array_CM update,              M_array_CM succ,
      int *newmaxlevel,               int k )
{
      int l;

      for (l=*newmaxlevel;l>Sfe->level;l--) update[l]=Sfe->header;
 /* The rows of the update array greathers than level(S) are initialized */
      for (l=1;l<=*newmaxlevel;l++)
            merge_level_CM(S,l,level,newnode,succ[l],update[l],k);
/* The skiplist S is updated level by level sequentially */
      if (Sfe->level<*newmaxlevel) Sfe->level=*newmaxlevel;
 /* The level of the skiplist is updated */
}
```
⟨*merge_level_CM*⟩

⟨*merge_level_CM*⟩≡

```
void merge_level_CM(skiplist_CM *S,   int l,
  /* The level to be merged */
      int:packet_shape *level,        ref_node:packet_shape *newnode,
      int:packet_shape succ_row,      int:packet_shape update_row,
      int k )
{
  /* packet_shape is the current shape */

    where (*level>=l){
      where (pcoord(0)>0 && pcoord(0)<k+1){

      /* The forward in succ_array is the NIL succ_array node */
           where (succ_row==k+1)
              [*newnode-l]S->forward=[update_row-l]S->forward;

      /* The forward in succ_array  has the same update */
           where (succ_row<k+1 && [succ_row]update_row==update_row)
              [*newnode-l]S->forward=[succ_row]*newnode;

      /* The forward in succ_array  has a different update */
           where (succ_row<k+1 && [succ_row]update_row!=update_row){
              [*newnode-l]S->forward=[update_row-l]S->forward;
              [[succ_row]update_row-l]S->forward=[succ_row]*newnode;
           }
      }
      where (pcoord(0)==[0]succ_row)[update_row-l]S->forward=*newnode;
    }
}
```

3.1.7 General procedures

We design the folowing procedures

⟨*implementation of CM general procedures*⟩≡
⟨*empty_skiplist_CM*⟩
⟨*spread_ref_node_CM*⟩
⟨*push_forward_CM*⟩
⟨*push_down_CM*⟩
⟨*split_CM*⟩
⟨*random_level_CM*⟩
⟨*log_CM*⟩
⟨*sort_CM*⟩

$\langle$*empty_skiplist_CM*$\rangle\equiv$

```
void empty_skiplist_CM ( skiplist_CM *S, skiplist_FE *Sfe)
{
  Sfe->header=number_nodelevels;
  Sfe->feet=1;
  Sfe->level=1;
  Sfe->number=0;
  [Sfe->header-1]S->forward=NIL;
  [Sfe->header-1]S->key=minkey;
  [NIL-1]S->key=maxkey;
}
```

$\langle$*spread_ref_node_CM*$\rangle\equiv$

```
ref_node:packet_shape spread_ref_node_CM(ref_node:packet_shape nl)
{
  bool:packet_shape mask;
  ref_node:packet_shape node;

  where (nl==0) mask=0;
  where (nl!=0) mask=nl;
  node=scan(nl,0,CMC_combiner_copy,CMC_upward,CMC_start_bit,&mask,
                 CMC_inclusive);
  return node;
}
```

$\langle$*push_forward_CM*$\rangle\equiv$

```
void push_forward_CM(set_of_packets *P, ref_node:packet_shape f)
{
  /* packet_shape is the current shape */
  P->node=f;
}
```

$\langle$*push_down_CM*$\rangle\equiv$

```
void push_down_CM(set_of_packets *P)
{
  /* packet_shape is the current shape */
  where (P->level==1) P->active=false;
    else  P->level--;
}
```

⟨*split_CM*⟩≡

```
void split_CM(set_of_packets *P)
{
  ref_node:packet_shape m;

  /* packet_shape is the current shape */

  m=P->first;                   m=m+P->last;
  m=m>>1;                       [m+1](P->first)=m+1;
  [m+1](P->last)=P->last;       [m+1](P->active)=true;
  [m+1](P->level)= P->level;    [m+1](P->node)= P->node;
  [P->first](P->last)=m;
}
```

⟨*random_level_CM*⟩≡

```
int:current random_level_CM(float p)
{
   int:current l,b;

   b=1;l=1;
   while (|=b)
     where (b)
       where (prand()%%10>=(int)(p*10))  l++;
       else b=0;
   return l;
}
```

⟨*sort_CM*⟩≡

```
int:current sort_CM(int:current a)
{
  int:current b;
  b=rank(a,0,CMC_upward,CMC_none,CMC_no_field);
  [b]a=a;
  return(a);
}
```

⟨*log_CM*⟩≡

```
int log_CM(int n)
{
  int l;
  l=0;
  while (n>1) {l++;n=n/2;}
  return l;
}
```

3.1.8 Specification of CM procedures

⟨*specification of CM procedures*⟩≡
  ⟨*specification of CM search procedures*⟩
  ⟨*specification of CM insert procedures*⟩
  ⟨*specification of CM general procedures*⟩

where

⟨*specification of CM search procedures*⟩≡
```
 void search_CM( key_type:packet_shape *,
      ref_key ,skiplist_CM * ,       skiplist_FE ,
      ref_node:packet_shape *);

void init_search_CM(key_type:packet_shape ,
      ref_key ,       skiplist_FE ,   set_of_packets *);

void route_CM ( set_of_packets *,     skiplist_CM *);
```

⟨*specification of CM insert procedures*⟩≡
```
void insert_CM(key_type:packet_shape *,       ref_key ,
               skiplist_CM *,                 skiplist_FE *);

void search_with_update_CM (key_type:packet_shape *,
      ref_key ,               skiplist_CM * ,
      skiplist_FE *,          ref_node:packet_shape *,
      M_array_CM );

void route_with_update_CM ( set_of_packets *P,
      skiplist_CM *S,         M_array_CM update);

void create_new_node_CM(key_type:packet_shape *,
      skiplist_CM *,          skiplist_FE *,
      ref_node:packet_shape *,int:packet_shape *,
      int *,                  int );

void make_succ_CM( int:packet_shape *,
      M_array_CM ,            int );

void merge_CM (skiplist_CM *, skiplist_FE *,
      ref_node:packet_shape *,int:packet_shape *,
      M_array_CM,             M_array_CM,
      int *,                  int );

void merge_level_CM(skiplist_CM *,
      int ,                   int:packet_shape *,
      ref_node:packet_shape *,int:packet_shape ,
      int:packet_shape,       int);
```

⟨*specification of CM general procedures*⟩≡

```
void empty_skiplist_CM ( skiplist_CM *, skiplist_FE *);
ref_node:packet_shape spread_ref_node_CM(ref_node:packet_shape );
void push_forward_CM(set_of_packets *, ref_node:packet_shape);
void push_down_CM(set_of_packets *);
void split_CM(set_of_packets *);
int:current random_level_CM(float p);
int:current sort_CM(int:current);
int log_CM(int);
```

3.1.9 Remainder code

The `main` program creates a skip list with `number_initial_insertions` keys whose values are $1000, 2000, \cdots, number\_initial\_insertions * 1000$. After a `number_of_insertions` keys are inserted.

⟨*main program*⟩≡

```
main ()
{
      skiplist_CM S;
      skiplist_FE Sfe;
      key_type:packet_shape a;

  /* The skiplist is created in parallel */
      empty_skiplist_CM(&S,&Sfe);
      with (packet_shape)  {
        a=pcoord(0)*1000;
         where (pcoord(0)<number_initial_insertions+2) {
           insert_CM(&a,number_initial_insertions,&S,&Sfe);
         }
        a=prand()%% maxkey;
        where (pcoord(0)<number_of_insertions+2){
               a=sort_CM(a);
          /* The skip list is updated in parallel */
               insert_CM(&a,number_of_insertions,&S,&Sfe);
        }
      }
}
```

In order to execute the program we need some external libreries

⟨*includes*⟩≡

```
#include <cscomm.h>
#include <stdio.h>
#include <stdlib.h>
```

2.306	2.160	2.060	2.081	2.080	2.055	2.027	2.118	2.046	2.017
2.005	2.141	2.181	2.161	2.131	2.139	2.039	2.042	2.014	2.035
2.014	2.069	2.153	2.116	2.105	2.179	2.167	2.026	2.049	2.018
2.029	2.033	2.061	2.842	2.974	2.151	2.019	2.185	2.167	2.028
2.009	2.130	2.049	2.020	2.006	2.148	2.027	1.956	2.054	2.169
2.430	2.026	2.204	2.038	2.131	1.977	2.027	2.029	2.161	2.159
2.324	2.181	2.048	2.030	2.137	2.042	2.018	2.019	2.030	1.997
2.057	2.047	2.046	1.998	2.016	2.027	2.036	2.038	2.149	2.046
2.124	2.033	2.066	2.018	2.174	2.011	2.014	2.107	2.191	2.413
2.038	2.030	1.996	2.053	2.026	2.038	2.026	2.026	2.015	2.048

Table 5: A sample of the time spent by the FE

1.053	0.953	1.053	0.953	1.053	1.149	1.053	1.050	1.039	1.048
1.046	1.157	1.555	1.257	1.453	1.252	1.141	1.050	1.053	1.149
1.051	1.055	0.954	1.356	1.154	1.454	1.152	1.053	1.046	1.047
1.052	1.148	1.148	1.155	1.252	1.152	1.152	1.153	1.157	1.151
0.953	1.354	1.050	1.154	1.052	1.052	1.042	0.953	1.355	1.254
1.353	1.049	1.155	1.148	1.053	1.063	1.152	1.149	1.152	1.252
1.254	1.256	1.042	1.054	1.052	1.355	1.147	1.050	1.149	1.147
1.253	1.153	1.151	1.253	1.243	1.251	1.245	1.149	1.155	1.245
1.153	1.054	1.052	1.052	1.353	0.954	1.150	1.447	1.252	0.953
1.149	1.052	1.050	1.053	0.953	1.047	1.149	1.147	1.052	1.250

Table 6: A sample of the time spent by the CM

Although the CM has 2024 processors with only 512 availables, the minimum declared length is should be 4096.

⟨*defines*⟩≡

```
#define number_nodelevels 32768
#define MAXk   number_nodelevels
#define M 20

#define NIL 1
#define minkey 0
#define maxkey number_nodelevels*1000

#define false 0
#define true 1

#define number_initial_insertions 16000
#define number_of_insertions 12000
```

	Average	Variance
FE	2.09706	0.02054
CM	1.1403	0.014877
FE+CM	3.23736	0.0438411

Table 7: Statistical magnitudes of the previous samples

Size of shape	4096	8192	16384	32768	65536	131072	262144
Time of FE	1.307	1.327	1.490	1.722	1.891	2.275	3.165
Time of CM	0.0912	0.133	0.249	0.533	0.931	1.879	3.775
Total time	1.398	1.460	1.739	2.256	2.822	4.154	6.940

Table 8: Insertions in shapes with different size

3.2 Performace of parallel algorithm

3.2.1 The accuracy of the data

The time sharing forces us to work with samples. For instance, the table 5 (table 6) contains the time spent in the Front End (Connection Machine) by the program that

- allocates an empty skip list,
- inserts 16000 keys numbered from 1 to 16000,
- inserts 16000 random keys.

The mean statistical magnitudes are shown in table 7. Thus we can work with samples of size 25 because the relative error will be smaller than the 5% in the CM an the 3% in the FE.

3.2.2 Size of the shapes

We analyse in this section the relation between the length of the shapes of the data structures and the running time. Althought the hardware contains small number of phisical processors, we can work with many virtual processors. This fact implies an increment of the time spent

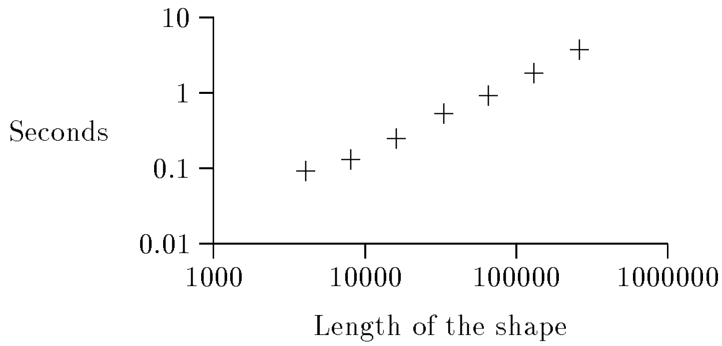

Figure 4: Running time with differents shpes

n	32000		8000		1000		250	
k	FE	CM	FE	CM	FE	CM	FE	CM
0	0.460	0.482	0.467	0.488	0.386	0.437	0.333	0.377
1	0.688	0.696	0.753	0.747	0.598	0.628	0.414	0.460
2	0.977	0.946	1.026	1.013	0.851	0.875	0.417	0.460
4	0.938	0.906	0.996	0.955	0.766	0.827	0.647	0.673
8	1.020	1.005	1.158	1.116	0.857	0.896	0.719	0.746
16	1.401	1.353	1.322	1.320	1.011	1.058	0.878	0.903
32	1.651	1.617	1.654	1.617	1.312	1.349	1.134	1.163
64	1.872	1.840	1.848	1.833	1.568	1.606	1.268	1.301
125	2.126	2.112	2.102	2.103	1.724	1.786	1.431	1.485
250	2.606	2.651	2.532	2.568	2.027	2.144	1.673	1.769
500	2.831	2.890	2.786	2.842	2.182	2.389		
1000	3.046	3.124	2.808	3.024	2.343	2.531		
2000	3.140	3.253	3.039	3.162				
4000	3.272	3.412	3.236	3.354				
8000	3.442	3.621	3.446	3.592				
16000	3.574	3.771						
32000	3.694	3.880						

Table 9: Time spent to insert k items

in the program, because each physical processor supports many virtual ones. For instance, the table 8 contains the time (seconds) spent to insert 1000 items in an skip list with 1000 items by using severals shapes.

The figure 4 represents the time spent by the Connection Machine. Recall that only 512 physical processors are availables, then each one has work as a $8, 16, 32, \cdots, 512$ virtual ones. The linearity of the representation confirms the proporcionality between the running time and the number of virtual processors.

3.2.3 The data

In order to analyse the performance of the insertion algorithm we have executed the program for several input data. The size of the shape was 65536. The routine considered was the *insert_CM* one, and we are keep the time spent by this routine on the Front End and CM. The table 9 and the figure 5 shows the data obtained

4 Race performance between both paradigms

The table 10 and the figure 6 contains the running times of the insertion of k items in a skip list of size 32000 .The parallel data are extracted from the third column of the table 9 (the time spent in the CM), and the sequential data are calculated from the approximation formula $t_n = 2.07310^{-5} n \log n$.

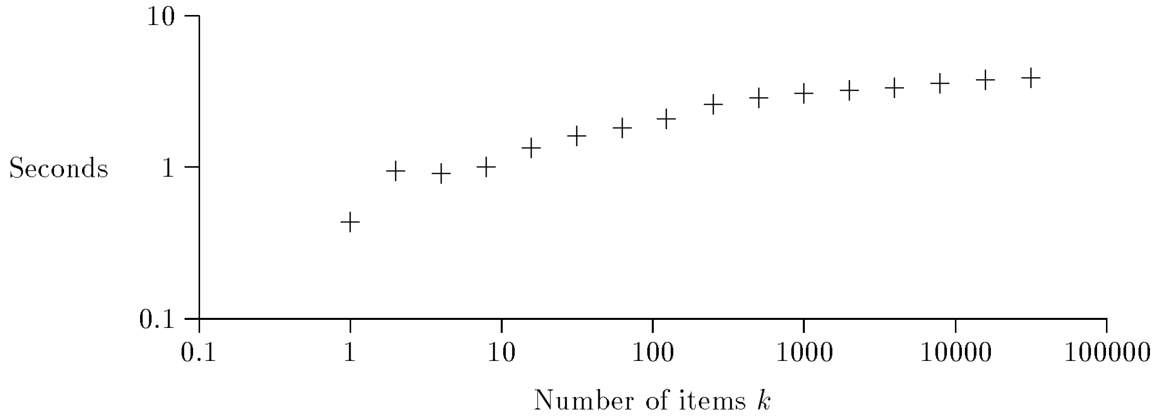

Figure 5: Running time of the *insert_CM* procedure

n	32000	
k	parallel	sequential
1	0.696	0.00034
2	0.946	0.00068
4	0.906	0.0013
8	1.005	0.0027
16	1.353	0.0054
32	1.617	0.011
64	1.840	0.022
125	2.112	0.043
250	2.651	0.085
500	2.890	0.17
1000	3.124	0.34
2000	3.253	0.68
4000	3.412	1.37
8000	3.621	2.75
16000	3.77	5.55
32000	3.880	11.25

Table 10: Comparation of the running time of the *insertCM* procedure

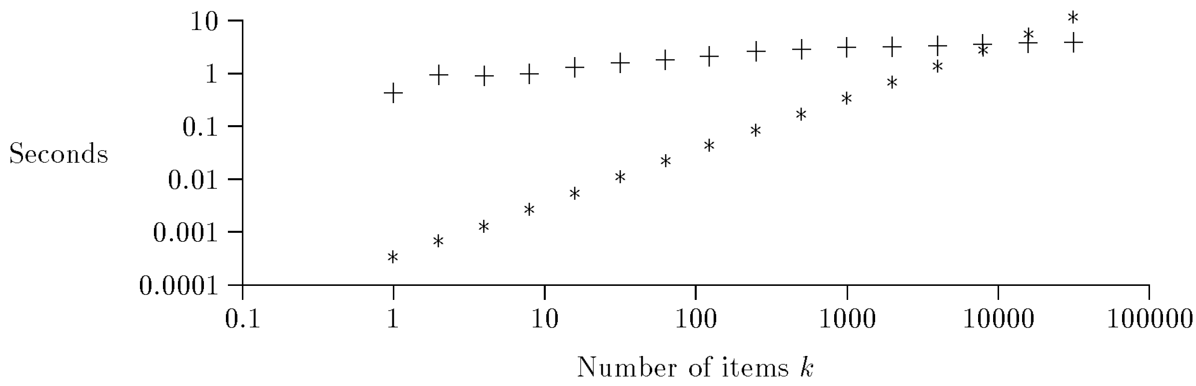

Figure 6: Comparation of the running time of the *insert_CM* procedure