APPEND

Asynchronous Parallel Data Structures in Fork95

Christoph W. Keßler **Oliver Fritzen**

FB IV - Informatik, Universität Trier, 54286 Trier, Germany
`kessler@psi.uni-trier.de`

April 7, 1999

APPEND (Asynchronous Parallel Programming Environment for Non-static Data structures) is a library of basic parallel data structures like hashtables, trees, dictionaries, priority queues, skip lists etc. written in Fork95. It is a parallel equivalent of LEDA focusing mainly on the asynchronous case, in contrast to J. Träff's PAD library that mainly covers synchronous parallel algorithms and data structures.

Contents

1 `PQueue`: A Parallel FIFO Queue

1.1 Definition

A k–way parallel FIFO queue is a data structure that distributes the queued items across k disjoint sublists that are accessed by both `put` and `get` in a round-robbin way. Hence, up to k accesses can perform in parallel. See [Roehrig '95] or [Keller,Kessler,Träff '99] for a detailed description.

1.2 Creation

`PQItem new_PQItem( void *`*data* `)`

generates a new `PQItem` containing a generic `data` pointer. A `PQItem` can be freed by

`void free_PQItem( PQItem` e `)`

Furthermore,

`PQueue new_PQueue( int` k `)`

creates a new k–way parallel FIFO queue. For performance reasons, k should be a power of 2.

1.3 Operations

- `void (*print)( PQueue` q `)`

  prints the current contents of q to standard output. `print` should be called by one processor only. Note that `print` is not protected against concurrent accesses of q.

- `int (*empty)( PQueue` q `)`

  returns a nonzero value if q is empty, and zero otherwise.

- `int (*size)( PQueue` q `)`

  returns the current number of elements in q, including the number of pending `put` operations minus the number of pending `get` operations.

- `PQueue (*put)( PQueue` q`, void *`x `)`

  appends the data item x to q. The `put` operation blocks until the appending is finished.

- `void *(*get)( PQueue` q `)`

  dequeues a data item from q and returns it. If q is empty, a NULL pointer is returned. Otherwise, the `get` operation blocks until the dequeueing is finished.

2 Parallel Hashtables

2.1 `HashTable`: A Parallel Rehashable Hashtable

2.1.1 Definition

The data type `HashTable` implements a hashtable. An instance H of `HashTable` stores `void` pointers to (furthermore unspecified) data elements.

The hashtable mainly consists of an array of list headers. The list elements, of type `HashTableItem`, consist of a pointer to the next list element and one pointer to the contained data element.

The *size* of the hashtable is the number of list headers in the array; the *length* of a list is the number of list elements in that list. A hashtable where all list headers are NULL is called *empty*.

If the lists containing the data items grow too big, a rehash with a larger hash table size (and, perhaps a better hash function) should be considered.

The *hash function* maps a data item referenced by a void pointer to an integer. It is to be supplied by the user and passed as a parameter to the constructor function of the hashtable.

Some operations require an additional parameter: *equals* denotes a pointer to a user-defined, integer-valued comparison function that takes two data pointers as parameters and returns 0 if it considers those data equal, and nonzero otherwise.

2.1.2 Creation

`Hashtable` H `= new_HashTable( int (`*hashFn*`)(void *), int` *Size*`)`

creates a `HashTable` instance with size *Size* and hash function *hashFn*.

2.1.3 Operations

- `void HashTableEnter( HashTable *`H`, void *`*entry*`)`

  inserts data item referenced with *entry* into hashtable.

- `void HashTableSingleEnter( HashTable *H, void *`*entry*`, int (*`*equals*`)(void *,void *))`

  same as `HashTableEnter()`, except that only those items are inserted into the hash table that aren't already in it. The meaning of equality of data items must be provided by the user-supplied function pointer *equals*.

- `void *HashTableLookup( HashTable *`H`, void *`*entry*`, int (*`*equals*`)(void *, void *))`

  Looks if data equal to **entry* is in the hash table H; if so, returns pointer to data, otherwise it returns NULL. The hash table remains unchanged. If *equals* matches several entries in H, then only one of these elements is returned.

- `void *HashTableExtract( HashTable *`H`, void *`*entry*`, int (*`*equals*`)(void *, void *))`

  Looks if data **entry* is in the hash table H; if so, removes data from hashtable and returns pointer to data; otherwise it returns NULL. If *equals* matches several entries in H, then only one of these elements is extracted.

- `sync void *HashTableRehash( HashTable *`H`, int (*`*newHashFn*`)(void *), int` *newSize*`)` reallocates the header array in hash table H with size *newSize* and rehashes all data contained in hashtable H according to the new hash function *newHashFn*. [1] Note that this function stands in "sync"-context.

- `int HashTableSize( HashTable *`H`)`

  Returns current size of hashtable H.

- `int HashTableNumElts( HashTable *`H`)`

  Returns number of data elements currently stored in the hash table H.

- `int HashTableListLength( HashTable *`H`, int `i`)`

  Returns number of data items in list with hashvalue i.

[1] Rehash uses a reader-writer-lock (with Rehash as writer and all other hash-changing operations as readers) to ensure that no concurrent rehashing can take place

2.2 `hashtable`: A Simple Parallel Hashtable

2.2.1 Definition

`hashtable` is a downgrade from the previously defined `HashTable` data structure. It features no rehashing and keeps no list length variables. All other functions and variables do the same as their counterparts in `HashTable`.

The list items used by `hashtable` are identical to these used by `HashTable`, namely `HashTableItem`.

`hashtable` avoids the performance penalty incurred by maintaining the option of rehashing in `HashTable`: Rehashing demands explicit exclusion of reader-writer-conflicts during rehashing; due to this, all critical `HashTable` operations are protected by reader-writer locks, which are relatively expensive. Hence, if no rehashing is required, `hashtable` should be preferred over `HashTable`.

2.2.2 Creation

`hashtable` *H* `= new_Hashtable( int (`*hashFn*`)(void *), int` *Size*`)`

creates an instance of `hashtable` with size *Size* and hash function *hashFn*.

2.2.3 Operations

- `void hashtableEnter( hashtable *`*H*`, void *`*entry*`)`

  inserts data item referenced by *entry* into hash table H.

- `void hashtableSingleEnter( hashtable *`*H*`, void *`*entry*`, int (*`*equals*`)(void *,void *))`

  same as `hashtableEnter()`, except that only those items are inserted into the hashtable H that aren't already in H.

- `void *hashtableLookup ( hashtable *`*H*`, void *`*entry*`, int (*`*equals*`)(void *, void *))`

  Looks if data *entry is in hashtable; if so, returns pointer to data, otherwise it returns null; hashtable remains unchanged. If *equals* matches several entries in H, then only one of these entries is returned.

- `void *hashtableExtract( hashtable *`*H*`, void *`*entry*`, int (*`*equals*`)(void *, void *))`

  Looks if data **entry* is in hash table H; if so, removes it from H and returns pointer to data; otherwise it returns null. If *equals* matches several entries in H, then only one of these elements is extracted.

- `int hashtableSize(hashtable *`*H*`)`

  Returns current size of hashtable H.

- `int hashtableNumElts(hashtable *`*H*`)`

  Returns number of data elements currently situated in the hashtable.

3 PSkipList: A Parallel Skip List

The data type `PSkipList` with its item type `PSkipListItem` implements a parallel skip list in Fork95. The implementation has been contributed by Christoph W. Keßler in March 1999.

As bibliographical references we recommend:

- W. Pugh, Communications of the ACM 1990.
- M. Weiss, Data Structures and Algorithm Analysis, 2nd ed., Benjamin-Cummings, 1994.

In order to use the implementation, the header file `<../util/skip.h>` must be included. It contains the necessary type definitions and function prototypes. Also, the user program must be linked with the object file `/util/skip.o`.

The implementation is generic for user–specified `Key` and `Inf` data types.

```
typedef void *Key;
typedef void *Inf;
```

The comparison function for `Key`s must be supplied by the user when generating an instance of the `PSkipList` data type.

Note that, due to its available operations, this parallel skip list may be used as

- a parallel dictionary,
- a parallel sorted sequence, and as
- a parallel priority queue.

Note that a sequence of concurrent `insert`, `delete`, `deleteMin` or `decreaseKey` operations needs not be committed to the skip list in the same order of time as the operations were called. If the user must guarantee that a certain set of operations is finished before another one can be started, she should put a `barrier` statement in between.

3.1 `PSkipListItem` data type

3.1.1 Definition

The data type `PSkipListItem` is used to store an item in a parallel skip list. It mainly contains a key entry that is used to compare and retrieve items, and an information entry that contains non–key data.

It has the following programmer interface:

```
typedef struct skiplistnode {
   Key key;    // points to a key element
   Inf inf;    // points to a data element
   // ... plus some hidden fields
} *PSkipListItem;
```

3.1.2 Creation

The *constructor* for a new skip list item

`PSkipListItem new_PSkipListItem( int` h`, Key` k `);`

takes as parameters the skip list node height h and the key entry k. The `inf` entry may be later set separately.

There is also a destructor available:

`void PSkipListItemFree( PSkipListItem` y `);`

For debugging purposes, the following routine can be used to print the current state of a skip list item y to standard output:

`void PSkipListItemPrint( PSkipListItem` y `);`

3.1.3 Implementation note

The memory to store a `PSkipListItem` instance is allocated on the permanent shared heap.

3.2 `PSkipList` data type

3.2.1 Definition

The skip list consists of linked nodes whose height is between 1 and some maximum height m specified by the user. The maximum height m should be approximately the floor of the base 2 logarithm of the average number of elements to be stored. It must not exceed the limit `PSKIPLISTMAXHEIGHT` predefined in `skip.h` (if necessary, modify this limit appropriately).

3.2.2 Creation

The constructor for an instance of `PSkipList`

`PSkipList new_PSkipList( int (*`*cmp*`)(Key,Key), int` m`, Key` *minKey* `)`

takes three parameters: the compare function for `Key`s, the maximum node height, and a very small `Key` value that must be smaller than the key of any element that may ever be inserted, looked up, or deleted from the skip list.

3.2.3 Operations

We use an OO-like programmer interface based on function pointers for the important operations on parallel skip lists. Because C (and thus Fork95) is not an OO language, we simulate this by passing the `this` object explicitly as first parameter. For instance,

$l \rightarrow$`print(` l `);`

prints the current contents of l to standard output.

On a `PSkipList` instance l, the following operations are possible:

- `void (*print)( PSkipList` l `)`

  prints the current contents of l to standard output.

- `int (*empty)( PSkipList` l `)`

  returns a nonzero value if l contains at least one element, and zero otherwise.

- `int (*size)( PSkipList` l `)`

  returns the current number of elements in l, including the number of pending `insert` operations.

- `PSkipListItem (*insert)( PSkipList` l`, Key` k`, Inf` i `)`

  inserts the item (k,i) into l. If there exists already an item with key k in l, the new item is inserted before it.

- `PSkipListItem (*locate)( PSkipList` l`, Key` k `)`

  returns a pointer to an item (k,i) if there is one in l, and NULL otherwise.

- `Inf (*access)( PSkipList` l`, Key` k `)`

  like `locate`, but returns the `Inf` entry i only.

- `PSkipListItem (*delete)( PSkipList` l`, Key` k `)`

  deletes an item (k,i) from l and returns a pointer to it if there is one, and returns NULL otherwise. If there are several items with key k in l, only one of them is deleted from l.

- `PSkipListItem (*deleteMin)( PSkipList` l `)`

  deletes the item from l that currently has the minimum key value. If l is empty, `deleteMin` returns NULL. If there exist several items with same minimum key value, `deleteMin()` deletes only one of them.

- `sync int (*deleteMins)( PSkipList` l`, int` n`, PSkipListItem *`a `)`

  deletes the n items that currently have the n smallest key values in l, assigns pointers to these items in the item array pointed to by a, and returns the number r of found (and assigned) minimum entries. If l contains only $r < n$ elements, all of them will be assigned. If several `deleteMins()` operations are in progress concurrently, atomicity of these operations is not guaranteed (i.e. the operations may report non-contiguous sequences of minimum elements).

- `PSkipListItem (*findMin)( PSkipList` l `)`

  returns a pointer to the item that currently has minimum key value in l. If l is empty, `findMin` returns NULL.

- `PSkipListItem (*pred)( PSkipList` l`, Key` k `)`

  returns a pointer to the item (k', i') in l with next smaller key k', i.e. k' is maximal with $k' < k$. If there are several items with key k' in l, a pointer to the first of them is returned. If k is the element with minimum key in l, a NULL pointer is returned.

- `PSkipListItem (*decreaseKey)( PSkipList` l`, Key` k`, Key` k' `)`

  changes the `Key` value k of an item (k, i) in l to the smaller value k', and returns a pointer to the modified item (k', i). If several items with same key k are in l, the operation is applied only to the first of them. If no key value k exists in l, or if $k' \geq k$, the operation returns NULL.

- `PSkipListItem (*changeInf)( PSkipList` l`, Key` k`, Inf` i `)`

  changes the `Inf` entry of an item with key k in l to the new value i.

3.2.4 Implementation notes

The implementation uses RWD-locks (see `lib/async.c`). The order of locking of nodes proceeds strictly from the "left" to the "right", i.e. follows the total order induced by the `next[0]` pointers. Hence, the implementation is deadlock–free.

Inserting or deleting a "high" node limits simultaneous access to the data structure much more than for a "small" node. By random coin tossing, the expected number of nodes of height h is 2^{-h}, $1 \leq h < m$.

By toggling the value of the preprocessor variable `SILENCE` in `util/skip.h` (and recompiling), the operations on the skip list can be traced.

The implementations of the following operations are left to the reader as an exercise:

```
PSkipListItem (*deleteMax)( PSkipList );
PSkipListItem (*findMax)( PSkipList );
PSkipListItem (*succ)( PSkipList, Key );
PSkipListItem (*locatePred)( PSkipList );
PSkipListItem (*locateSucc)( PSkipList );
void          (*split)( PSkipList, PSkipList *, PSkipList *);
void          (*concat)( PSkipList, PSkipList );
```