Frequent Pattern-Growth with MPI

Generated by Doxygen 1.8.17

1 File Index

1.1 File List

Here is a list of all files with brief descriptions:

2 File Documentation

2.1 src/io.h File Reference

Functions that handle the input and output of the program.

```
#include "types.h"
#include <mpi.h>
```

Functions

- void update_supports (Item item, SupportMap *support_map)

  Increase the support of the given item in the map. If the item is not present it is inserted with support 1.
- void transaction_free (Transaction *transaction)

  Free the given transaction and all the items in it.
- void transactions_free (TransactionsList *transactions)

  Free all the transaction in the list and the list itself.
- void transactions_write (int rank, TransactionsList transactions)

  Write a list of transactions to the file named as the rank of the process.
- int item_parse (int rank, int i, char *chunk, int chunk_size, Transaction *transaction, SupportMap *support_map)

  Parse an item from the string chunk, starting from position i up to the first space, newline or '\0'. The item is added to the transaction and the corresponding support is increased.
- int transaction_parse (int rank, int i, char *chunk, int chunk_size, TransactionsList *transactions, SupportMap *support_map)

  Parse an transaction from the string chunk, starting from position i up to the first newline or '\0'. The transaction is added to the list of transactions. The support of the items in the transaction is increased as they get read.
- void read_chunk (char *filename, int rank, int world_size, char **chunk, int *my_size, int *read_size)

  Read a chunk of the given file.
- void transactions_read (TransactionsList *transactions, char *filename, int rank, int world_size, SupportMap *support_map)

  Read a list of transactions from the portion of file assigned to the current process. The support of the items read is increased in the support_map.

2.1.1 Detailed Description

Functions that handle the input and output of the program.

2.1.2 Function Documentation

2.1.2.1 item_parse()

```
int item_parse (
        int rank,
        int i,
        char * chunk,
        int chunk_size,
        Transaction * transaction,
        SupportMap * support_map )
```

Parse an item from the string chunk, starting from position i up to the first space, newline or '\0'. The item is added to the transaction and the corresponding support is increased.

Parameters

rank	Rank of the current process
i	Start position from where to start parsing
chunk	String containing the item to parse
chunk_size	Size of the chunk
transaction	The transaction where to add the item
support_map	The map from items to respective support

Returns

The index where the parsed item ends (excluded)

2.1.2.2 read_chunk()

```
void read_chunk (
        char * filename,
        int rank,
        int world_size,
        char ** chunk,
        int * my_size,
        int * read_size )
```

Read a chunk of the given file.

Read up to 2 ∗ my_size, since we do not know where transactions start exactly.

Parameters

filename	File where transactions are stored
rank	Rank of the current process
world_size	Number of active processes
chunk	String where to store the read bytes
my_size	Number of bytes each process is assigned to
read_size	Number of bytes the current process has read

2.1.2.3 transaction_free()

```
void transaction_free (
    Transaction * transaction )
```

Free the given transaction and all the items in it.

Parameters

transaction	The transaction to free

2.1.2.4 transaction_parse()

```
int transaction_parse (
    int rank,
    int i,
    char * chunk,
    int chunk_size,
    TransactionsList * transactions,
    SupportMap * support_map )
```

Parse an transaction from the string chunk, starting from position i up to the first newline or '\0'. The transaction is added to the list of transactions. The support of the items in the transaction is increased as they get read.

Parameters

rank	Rank of the current process
i	Start position from where to start parsing
chunk	String containing the item to parse
chunk_size	Size of the chunk
transactions	The list of transactions where to add the transaction
support_map	The map from items to respective support

Returns

The index where the parsed transaction ends (excluded)

2.1.2.5 transactions_free()

```
void transactions_free (
        TransactionsList * transactions )
```

Free all the transaction in the list and the list itself.

Parameters

transactions	The list of transactions to free

2.1.2.6 transactions_read()

```
void transactions_read (
        TransactionsList * transactions,
        char * filename,
        int rank,
        int world_size,
        SupportMap * support_map )
```

Read a list of transactions from the portion of file assigned to the current process. The support of the items read is increased in the support_map.

Parameters

transactions	List of transactions where to store the data
filename	Name of the file from which to read
rank	Rank of the current process
world_size	Number of active processes
support_map	A map from items to the respective support

2.1.2.7 transactions_write()

```
void transactions_write (
        int rank,
        TransactionsList transactions )
```

Write a list of transactions to the file named as the rank of the process.

Parameters

rank	Rank of the current process
transactions	List of transactions to write

2.1.2.8 update_supports()

```
void update_supports (
        Item item,
        SupportMap * support_map )
```

Increase the support of the given item in the map. If the item is not present it is inserted with support 1.

Parameters

item	Item of wich to increase the support
support_map	A map from items to the respective support

2.2 src/reduce.h File Reference

Functions that perform the exchange of data between MPI processes.

```
#include "mpi.h"
#include "tree.h"
#include "types.h"
```

Functions

- MPI_Datatype define_datatype_hashmap_element ()

  Define a datatype for an hashmap element in order to be able to send it with MPI.
- MPI_Datatype define_datatype_tree_node ()

  Define an MPI Datatype for a TreeNode in order to be able to send it with MPI.
- void merge_map (SupportMap *support_map, hashmap_element *elements, int size)

  Merge the current map with an array of elements.
- void recv_map (int rank, int world_size, int source, SupportMap *support_map, MPI_↩ Datatype DT_HASHMAP_ELEMENT)

  Receive an array of hashmap elements with MPI.
- void send_map (int rank, int world_size, int dest, SupportMap *support_map, MPI_↩ Datatype DT_HASHMAP_ELEMENT)

  Send an array of hashmap elements with MPI.

- void broadcast_map (int rank, int world_size, SupportMap *support_map, hashmap_element **items_count, int *num_items, MPI_Datatype DT_HASHMAP_ELEMENT, int min_support)

  Broadcast all the elements of a SupportMap to every MPI process.
- void get_global_map (int rank, int world_size, SupportMap *support_map, hashmap_element **items_count, int *num_items, int min_support)

  Get the global map object for every MPI process.
- void merge_indices (int rank, int *sorted_indices, int start1, int end1, int start2, int end2, hashmap_element *items_count, int num_items)

  Merge two arrays of sorted indices into a unique array of sorted indices.
- void recv_indices (int rank, int source, int *sorted_indices, int start, int *end, int length, int size, hashmap_element *items_count, int num_items)

  Receive an array of sorted indices.
- void send_indices (int rank, int dest, int *sorted_indices, int start, int end, int length)

  Send an array of sorted indices.
- void broadcast_indices (int rank, int *sorted_indices, int num_items)

  Broadcast the final global array of indices to every MPI process in the world.
- void get_sorted_indices (int rank, int world_size, int *sorted_indices, int start, int end, int length, hashmap_element *items_count, int num_items)

  Get the global sorted indices array for every MPI process.
- void parse_tree (TreeNodeToSend *nodes, int num_nodes, Tree *dest)

  Parse an array of TreeNodesToSend into a Tree structure.
- void send_tree (int dest, Tree *tree, MPI_Datatype DT_TREE_NODE)

  Sends a tree to an MPI process and frees up the memory.
- void recv_tree (int source, Tree *tree, MPI_Datatype DT_TREE_NODE)

  Receive a tree from an MPI process.
- void broadcast_tree (int rank, Tree *tree, MPI_Datatype DT_TREE_NODE)

  Broadcast the final FP-Tree to every MPI process in the world.
- void get_global_tree (int rank, int world_size, Tree *tree)

  Get the global FP-tree on every MPI process.

2.2.1 Detailed Description

Functions that perform the exchange of data between MPI processes.

2.2.2 Function Documentation

2.2.2.1 broadcast_indices()

```
void broadcast_indices (
        int rank,
        int * sorted_indices,
        int num_items )
```

Broadcast the final global array of indices to every MPI process in the world.

Parameters

rank	The rank of the process that broadcasts the data
sorted_indices	The array of sorted indices that has to be broadcasted
num_items	The number of elements of the sorted_indices array

2.2.2.2 broadcast_map()

```
void broadcast_map (
    int rank,
    int world_size,
    SupportMap * support_map,
    hashmap_element ** items_count,
    int * num_items,
    MPI_Datatype DT_HASHMAP_ELEMENT,
    int min_support )
```

Broadcast all the elements of a SupportMap to every MPI process.

This is used by the master process to send the global version of the map to every MPI process in the world

Parameters

rank	The rank of the process that broadcasts the data
world_size	The number of MPI processes in the world
support_map	The map from which to extract the elements to be sent
items_count	A pointer to an array of hashmap elements having the item string as a key and the support count as a value
num_items	A pointer to an integer describing the total number of items in the broadcasted map
DT_HASHMAP_ELEMENT	An MPI datatype that describes the data structure that has to be sent with MPI
min_support	The mininum support that an item has to have in order to be contained in the final map

2.2.2.3 broadcast_tree()

```
void broadcast_tree (
    int rank,
    Tree * tree,
    MPI_Datatype DT_TREE_NODE )
```

Broadcast the final FP-Tree to every MPI process in the world.

Parameters

rank	The rank of the process that broadcasts the data
tree	The tree that has to be broadcasted in case the process rank is 0, otherwise the tree which the final tree will be stored
DT_TREE_NODE	MPI_Datatype describing a TreeNode

2.2.2.4 define_datatype_hashmap_element() `MPI_Datatype define_datatype_hashmap_element ( )`

Define a datatype for an hashmap element in order to be able to send it with MPI.

Returns

MPI_Datatype describing an hashmap element

2.2.2.5 define_datatype_tree_node() `MPI_Datatype define_datatype_tree_node ( )`

Define an MPI Datatype for a TreeNode in order to be able to send it with MPI.

Returns

MPI_Datatype describing a TreeNode

2.2.2.6 get_global_map()

```
void get_global_map (
        int rank,
        int world_size,
        SupportMap * support_map,
        hashmap_element ** items_count,
        int * num_items,
        int min_support )
```

Get the global map object for every MPI process.

This function follows a tree-like structure to build the global map of items of our program. Initially, every process has its own partial map of items. Then, processes which meet the condition (rank % pow != 0), where pow is a function of the current level in the tree, sends its partial map to the specified destination. On the contrary, processes which meet the condition (rank % pow == 0) receive the map from the other processes, and then merge the received elements with their current map. This is done until the only process in the current level is the process number 0. Upon reaching that state, it means that process 0 now has the complete map of every process and can broadcast its knowledge to the whole domain.

Parameters

rank	The rank of the process that broadcasts the data
world_size	The number of MPI processes in the world
support_map	The map from which to extract the elements to be sent
items_count	A pointer to an array of hashmap elements having the item string as a key and the support count as a value
num_items	A pointer to an integer describing the total number of items in the broadcasted map
min_support	The mininum support that an item has to have in order to be contained in the final map

2.2.2.7 get_global_tree()

```
void get_global_tree (
        int rank,
        int world_size,
        Tree * tree )
```

Get the global FP-tree on every MPI process.

This function follows a tree-like structure to build the final FP-tree. Initially, every process has its own partial tree composed by the transactions of its local assigned chunk. Then, processes which meet the condition (rank % pow != 0), where pow is a function of the current level in the tree, sends its partial tree to the specified destination. On the contrary, processes which meet the condition (rank % pow == 0) receive the FP-tree from the other processes, and then merge the received partial FP-tree with their current FP-tree. This is done until the only process in the current level is the process number 0. Upon reaching that state, it means that process 0 now has the complete FP-tree and can broadcast its knowledge to the whole domain.

Parameters

rank	The rank of the process that broadcasts the data
world_size	The number of processes in the current world
tree	The tree that has to be sent/received. This structure is heavily manipulated during the execution of this function.

2.2.2.8 get_sorted_indices()

```
void get_sorted_indices (
        int rank,
        int world_size,
        int * sorted_indices,
        int start,
        int end,
```

```
int length,
hashmap_element * items_count,
int num_items )
```

Get the global sorted indices array for every MPI process.

This function follows a tree-like structure to build the global sorted array of indices of the items of our program. Initially, every process has its own partial array of sorted indices. Then, processes which meet the condition (rank % pow != 0), where pow is a function of the current level in the tree, sends its partial array to the specified destination. On the contrary, processes which meet the condition (rank % pow == 0) receive the array from the other processes, and then merge the received elements with their current array. This is done until the only process in the current level is the process number 0. Upon reaching that state, it means that process 0 now has the complete array of every process and can broadcast its knowledge to the whole domain.

Parameters

rank	The rank of the process that broadcasts the data
world_size	The number of MPI processes in the current world
sorted_indices	The array of sorted indices that will be used to send the data
start	The starting index of the items that the function will send
end	The ending index of the items that the function will send
length	The length of each process' original interval width
items_count	An array of hashmap elements having the item string as a key and the support count as a value
num_items	The number of items in the sorted_indices array

2.2.2.9 merge_indices()

```
void merge_indices (
int rank,
int * sorted_indices,
int start1,
int end1,
int start2,
int end2,
hashmap_element * items_count,
int num_items )
```

Merge two arrays of sorted indices into a unique array of sorted indices.

Parameters

rank	The rank of the process that broadcasts the data
sorted_indices	The array of sorted indices that contains both arrays that have to be sorted
start1	The starting index of the first array
end1	The ending index of the first array

Parameters

start2	The starting index of the second array
end2	The ending index of the second array
items_count	An array of hashmap elements having the item string as a key and the support count as a value
num_items	The number of items in the sorted_indices array

2.2.2.10 merge_map()

```
void merge_map (
    SupportMap * support_map,
    hashmap_element * elements,
    int size )
```

Merge the current map with an array of elements.

Parameters

support_map	A pointer to the map which will be populated with the merged items
elements	The array of hashmap elements to merge
size	The size of the elements array

2.2.2.11 parse_tree()

```
void parse_tree (
    TreeNodeToSend * nodes,
    int num_nodes,
    Tree * dest )
```

Parse an array of TreeNodesToSend into a Tree structure.

Parameters

nodes	The array of TreeNodesToSend that have to be parsed
num_nodes	The size of the nodes array
dest	A pointer to the tree that will be created

2.2.2.12 recv_indices()

```
void recv_indices (
    int rank,
    int source,
```

```
int * sorted_indices,
int start,
int * end,
int length,
int size,
hashmap_element * items_count,
int num_items )
```

Receive an array of sorted indices.

Parameters

rank	The rank of the process that broadcasts the data
source	The rank of the MPI process that sends the data
sorted_indices	The array of sorted indices where the received data will be stored
start	The starting index of the items that the function will receive
end	The ending index of the items that the function will receive
length	The length of each process' original interval width
size	The current size of the received data
items_count	An array of hashmap elements having the item string as a key and the support count as a value
num_items	The number of items in the sorted_indices array

2.2.2.13 recv_map() `void recv_map (`

```
int rank,
int world_size,
int source,
SupportMap * support_map,
MPI_Datatype DT_HASHMAP_ELEMENT )
```

Receive an array of hashmap elements with MPI.

First, receive the size of the array of hashmap elements and then receive the actual elements

Parameters

rank	MPI process rank
world_size	Number of MPI processes in the current world
source	The rank of the MPI process that sends the array of elements
support_map	A pointer to the map where the received elements have to be inserted
DT_HASHMAP_ELEMENT	An MPI datatype that describes the data structure received

2.2.2.14 recv_tree()

```
void recv_tree (
    int source,
    Tree * tree,
    MPI_Datatype DT_TREE_NODE )
```

Receive a tree from an MPI process.

Parameters

source	The MPI process that is sending the data
tree	A pointer to partial tree of the current process, which will be integrated by merging the received tree
DT_TREE_NODE	MPI_Datatype describing a TreeNode

2.2.2.15 send_indices()

```
void send_indices (
    int rank,
    int dest,
    int * sorted_indices,
    int start,
    int end,
    int length )
```

Send an array of sorted indices.

Parameters

rank	The rank of the process that broadcasts the data
dest	The rank of the MPI process that will receive the data
sorted_indices	The array of sorted indices that will be used to send the data
start	The starting index of the items that the function will send
end	The ending index of the items that the function will send
length	The length of each process' original interval width

2.2.2.16 send_map()

```
void send_map (
    int rank,
    int world_size,
    int dest,
    SupportMap * support_map,
    MPI_Datatype DT_HASHMAP_ELEMENT )
```

Send an array of hashmap elements with MPI.

First, send the size of the array of hashmap elements and then send the actual elements.

Parameters

rank	MPI process rank
world_size	Number of MPI processes in the current world
dest	The rank of the MPI process that will receive the array of elements
support_map	A pointer to the map where the elements to be sent are stored
DT_HASHMAP_ELEMENT	An MPI datatype that describes the data structure that has to be sent with MPI

2.2.2.17 send_tree()

```
void send_tree (
    int dest,
    Tree * tree,
    MPI_Datatype DT_TREE_NODE )
```

Sends a tree to an MPI process and frees up the memory.

Parameters

dest	The destination process that will receive the tree
tree	The tree that has to be sent
DT_TREE_NODE	MPI_Datatype describing a TreeNode

2.3 src/sort.h File Reference

Functions that implement parallel QuickSort with OpenMP.

```
#include "hashmap/hashmap.h"
#include "types.h"
```

Macros

- #define INSERTION_SORT_THRESH 100

Functions

- void insertion_sort (hashmap_element ∗items_count, int num_items, int ∗sorted_indices, int start, int end)

  Sort the indices contained in the array sorted_indices from start to end using insertion sort. The indices are sorted according to the corresponding element in items_count.

- void swap (int *a, int i, int j)

  Swap elements in position i and j in the array a.
- int pivot (hashmap_element *items_count, int num_items, int *sorted_indices, int start, int end, int m)

  Perform the pivot operation of the quicksort algorithm on the sub-array sorted_indices from start to end (included).
- int choose_pivot (hashmap_element *items_count, int num_items, int *sorted_indices, int start, int end)

  Choose a pivot according to the Median-of-three heuristic.
- void parallel_quick_sort (hashmap_element *items_count, int num_items, int *sorted_↩ indices, int start, int end, int **stack, int *num_busy_threads, int *num_threads)

  Parallel version of the QuickSort algoritm.
- void sort (hashmap_element *items_count, int num_items, int *sorted_indices, int start, int end, int num_threads)

  Put in the array sorted_indices the indices of the elements of the array items_count from position start to end, after they are sorted according to their value field.

2.3.1 Detailed Description

Functions that implement parallel QuickSort with OpenMP.

2.3.2 Macro Definition Documentation

2.3.2.1 INSERTION_SORT_THRESH `#define INSERTION_SORT_THRESH 100`

2.3.3 Function Documentation

2.3.3.1 choose_pivot()

```
int choose_pivot (
        hashmap_element * items_count,
        int num_items,
        int * sorted_indices,
        int start,
        int end )
```

Choose a pivot according to the Median-of-three heuristic.

The pivot is chosen as the median of the first, middle and last element of the sub-array sorted↩ _indices from start to end, according to the corresponding value in items_count. These three indices are swapped, so that the median at the end is in position start.

Parameters

items_count	array of key-value pairs
num_items	length of items_count
sorted_indices	array of indices that are going to be sorted
start	start position of the sub-array to sort
end	end position of the sub-array to sort

Returns

the position of the pivot

2.3.3.2 insertion_sort()

```
void insertion_sort (
        hashmap_element * items_count,
        int num_items,
        int * sorted_indices,
        int start,
        int end )
```

Sort the indices contained in the array sorted_indices from start to end using insertion sort. The indices are sorted according to the corresponding element in items_count.

Parameters

items_count	array of key-value pairs
num_items	length of items_count
sorted_indices	array of indices that are going to be sorted
start	start position of the sub-array to sort
end	end position of the sub-array to sort

2.3.3.3 parallel_quick_sort()

```
void parallel_quick_sort (
        hashmap_element * items_count,
        int num_items,
        int * sorted_indices,
        int start,
        int end,
        int ** stack,
        int * num_busy_threads,
        int * num_threads )
```

Parallel version of the QuickSort algoritm.

Parameters

items_count	array of key-value pairs
num_items	length of items_count
sorted_indices	array of indices that are going to be sorted
start	start position of the sub-array to sort
end	end position of the sub-array to sort
stack	stack containing the jobs that need to be done by free threads
num_busy_threads	Number of currently busy threads
num_threads	Number of threads that perform the sorting

2.3.3.4 pivot() `int pivot (`

```
hashmap_element * items_count,
int num_items,
int * sorted_indices,
int start,
int end,
int m )
```

Perform the pivot operation of the quicksort algorithm on the sub-array sorted_indices from start to end (included).

The indices which value in items_count is smaller than the value of the pivot are moved to the left of the pivot, the ones with a value greater or equal are moved to its right.

Parameters

items_count	array of key-value pairs
num_items	length of items_count
sorted_indices	array of indices that are going to be sorted
start	start position of the sub-array to sort
end	end position of the sub-array to sort
m	position of the pivot

Returns

the new position of the pivot

2.3.3.5 sort() `void sort (`

```
hashmap_element * items_count,
```

```
int num_items,
int * sorted_indices,
int start,
int end,
int num_threads )
```

Put in the array sorted_indices the indices of the elements of the array items_count from position start to end, after they are sorted according to their value field.

The array items_count is not modified. The algorithm implement a parallel version of QuickSort, described in the paper Süß M, Leopold C. A user's experience with parallel sorting and Open↩ MP. InProceedings of the Sixth European Workshop on OpenMP-EWOMP'04 2004 Oct 18 (pp. 23-38).

Parameters

items_count	array of key-value pairs
num_items	length of items_count
sorted_indices	array of indices that are going to be sorted
start	start position of the sub-array to sort
end	end position of the sub-array to sort
num_threads	Number of threads that perform the sorting

2.3.3.6 swap()

```
void swap (
int * a,
int i,
int j )
```

Swap elements in position i and j in the array a.

Parameters

a	array where to swap the elements
i	position of the first element to swap
j	position of the second element to swap

2.4 src/tree.h File Reference

Definition of FP-Tree and functions to build it.

```
#include "types.h"
```

Data Structures

- struct TreeNode

  A node of an FP-Tree. More...
- struct TreeNodeToSend

  A node of an FP-Tree to send with MPI. More...

Macros

- #define TREE_NODE_NULL -1

Typedefs

- typedef TreeNode ∗∗ Tree

  FP-Tree.

Functions

- TreeNode ∗ tree_node_new (int key, int value, int parent)

  Instantiate a new node of the tree.
- void tree_node_free (TreeNode ∗node)

  Free a tree node.
- Tree tree_new ()

  Instantiate a new tree.
- void tree_free (Tree ∗tree)

  Free the tree.
- int tree_add_node (Tree ∗tree, TreeNode ∗node)

  Add a node to the tree.
- void tree_add_subtree (Tree ∗dest, Tree source, int nd, int ns)

  Add the subtree rooted in the ns(th) node of the tree source as a child of the nd(th) node of the tree dest. The source tree is modified, as the nodes are moved to the destination tree.
- void tree_merge_dfs (Tree ∗dest, Tree source, int nd, int ns)

  Merge the subtree of dest rooted in node with id nd with the subtree of source rooted in ns and store the result in dest. Also the source tree is modified.
- void tree_merge (Tree ∗dest, Tree source)

  Merge the trees dest and source and store the result in dest. The source tree is modified. It is a wrapper for.
- void tree_get_nodes (Tree tree, TreeNodeToSend ∗∗nodes)

  Inserts into the vector nodes the nodes to send.
- void tree_print (Tree tree)

  Print the tree.
- Tree tree_build_from_transaction (int rank, int world_size, Transaction ∗transaction, IndexMap index_map, hashmap_element ∗items_count, int num_items, int ∗sorted_↩ indices)

  Build a tree given a transaction.
- Tree tree_build_from_transactions (int rank, int world_size, TransactionsList transactions, IndexMap index_map, hashmap_element ∗items_count, int num_items, int ∗sorted_↩ indices, int num_threads)

  Build a tree given a list of transactions.

2.4.1 Detailed Description

Definition of FP-Tree and functions to build it.

2.4.2 Data Structure Documentation

2.4.2.1 struct TreeNode A node of an FP-Tree.

Data Fields

map_t	adj	Adjacency map of the node, where the values are the indices of the children of the node in the tree and the keys are the ids of the corresponding items.
int	key	Id of the item represented by the node
int	parent	Index of the parent of the node in the tree
int	value	Value of the item represented by the node

2.4.2.2 struct TreeNodeToSend A node of an FP-Tree to send with MPI.

Data Fields

int	key	Id of the item represented by the node
int	parent	Index of the parent of the node in the tree
int	value	Value of the item represented by the node

2.4.3 Macro Definition Documentation

2.4.3.1 TREE_NODE_NULL `#define TREE_NODE_NULL -1`

2.4.4 Typedef Documentation

2.4.4.1 Tree `typedef TreeNode* * Tree`

FP-Tree.

2.4.5 Function Documentation

2.4.5.1 tree_add_node() `int tree_add_node (`

```
Tree * tree,
TreeNode * node )
```

Add a node to the tree.

Parameters

tree	Pointer to the tree
node	Pointer to the node to add

Returns

The id of the node in the tree

2.4.5.2 tree_add_subtree() `void tree_add_subtree (`

```
Tree * dest,
Tree source,
int nd,
int ns )
```

Add the subtree rooted in the ns(th) node of the tree source as a child of the nd(th) node of the tree dest. The source tree is modified, as the nodes are moved to the destination tree.

Parameters

dest	Pointer to the destination tree
source	Pointer to the source tree
nd	Id of the node in the destination tree
ns	Id of the node in the source tree

2.4.5.3 tree_build_from_transaction() `Tree tree_build_from_transaction (`

```
int rank,
int world_size,
Transaction * transaction,
IndexMap index_map,
hashmap_element * items_count,
int num_items,
int * sorted_indices )
```

Build a tree given a transaction.

Parameters

rank	The rank of the process
world_size	The number of processes in the world
transaction	The transaction
index_map	The map from item to the corresponding id
items_count	The array of hashmap elements having the item string as a key and the support count as a value
num_items	The number of items in the sorted_indices array
sorted_indices	The array of the sorted indices of the items

Returns

The built tree

2.4.5.4 tree_build_from_transactions()

```
Tree tree_build_from_transactions (
int rank,
int world_size,
TransactionsList transactions,
IndexMap index_map,
hashmap_element * items_count,
int num_items,
int * sorted_indices,
int num_threads )
```

Build a tree given a list of transactions.

First, we build the trees for the single transactions. Then, we merge them in a binary-tree-like fashion.

Parameters

rank	The rank of the process
world_size	The number of processes in the world
transactions	

Parameters

index_map	The map from item to the corresponding id
items_count	The array of hashmap elements having the item string as a key and the support count as a value
num_items	The number of items in the sorted_indices array
sorted_indices	The array of the sorted indices of the items
num_threads	The number of threads requested to perform the building

Returns

The built tree

2.4.5.5 tree_free() `void tree_free (`

```
Tree * tree )
```

Free the tree.

Parameters

tree	Pointer to the tree to free

2.4.5.6 tree_get_nodes() `void tree_get_nodes (`

```
Tree tree,
TreeNodeToSend ** nodes )
```

Inserts into the vector nodes the nodes to send.

Parameters

tree	The trees from which to get the nodes
nodes	The vector in which the nodes are put

2.4.5.7 tree_merge() `void tree_merge (`

```
Tree * dest,
Tree source )
```

Merge the trees dest and source and store the result in dest. The source tree is modified. It is a wrapper for.

See also

tree_merge_dfs()

Parameters

dest	The destination tree
source	The source tree

2.4.5.8 tree_merge_dfs()

```
void tree_merge_dfs (
        Tree * dest,
        Tree source,
        int nd,
        int ns )
```

Merge the subtree of dest rooted in node with id nd with the subtree of source rooted in ns and store the result in dest. Also the source tree is modified.

Parameters

dest	The destination tree
source	The source tree
nd	Id of the node in the destination tree
ns	Id of the node in the source tree

2.4.5.9 tree_new()

```
Tree tree_new ( )
```

Instantiate a new tree.

Returns

The new tree

2.4.5.10 tree_node_free()

```
void tree_node_free (
        TreeNode * node )
```

Free a tree node.

Parameters

node	Pointer to the node to free

2.4.5.11 tree_node_new()

```
TreeNode* tree_node_new (
        int key,
        int value,
        int parent )
```

Instantiate a new node of the tree.

Parameters

key	The key of the node
value	The value of the node
parent	The parent of the node in the tree

Returns

Pointer to the node created

2.4.5.12 tree_print()

```
void tree_print (
        Tree tree )
```

Print the tree.

Parameters

tree	The tree to print

2.5 src/types.h File Reference

Definition of datatypes used by the program.

```
#include "cvector/cvector.h"
#include "hashmap/hashmap.h"
#include <stdbool.h>
```

Typedefs

- typedef map_t SupportMap

  Map from Item to the corresponding support.
- typedef map_t IndexMap

  Map from Item to its id.
- typedef uint8_t * Item

  Item of a transaction.
- typedef Item * Transaction

  Transaction of items.
- typedef Transaction * TransactionsList

  List of transactions.

2.5.1 Detailed Description

Definition of datatypes used by the program.

2.5.2 Typedef Documentation

2.5.2.1 IndexMap `typedef map_t IndexMap`

Map from Item to its id.

2.5.2.2 Item `typedef uint8_t* Item`

Item of a transaction.

2.5.2.3 SupportMap `typedef map_t SupportMap`

Map from Item to the corresponding support.

2.5.2.4 Transaction `typedef Item* Transaction`

Transaction of items.

2.5.2.5 TransactionsList `typedef Transaction* TransactionsList`

List of transactions.

2.6 src/utils.h File Reference

Utility functions.

Functions

- int min (int a, int b)

  Minimum between two integers.
- int max (int a, int b)

  Maximum between two integers.
- int ulength (uint8_t *s)

  Length of an Item.

2.6.1 Detailed Description

Utility functions.

2.6.2 Function Documentation

2.6.2.1 max() `int max ( int a, int b )`

Maximum between two integers.

Parameters

a	first parameter
b	second parameter

Returns

maximum between a and b

2.6.2.2 min() `int min ( int a, int b )`

Minimum between two integers.

Parameters

a	first parameter
b	second parameter

Returns

minimum between a and b

2.6.2.3 ulength() `int ulength ( uint8_t * s )`

Length of an Item.

Parameters

s	Item

Returns

Number of characters in s