Journal of
Information Technology & Software Engineering

Feng et al. J Inform Tech Soft Engg 2011, 1:1
DOI: 10.4172/2165-7866.1000103

Research Article Open Access

A Parallelized Binary Search Tree

Jian Feng[1], Daniel Q. Naiman[2] and Bret Cooper[3]*

[1]Snoqualmie, WA 98065
[2]Department of Applied Mathematics and Statistics, The Johns Hopkins University, Baltimore, MD 21218, USA
[3]Soybean Genomics and Improvement Laboratory, USDA-ARS, Beltsville, MD 20705, USA

Abstract

PTTRNFNDR is an unsupervised statistical learning algorithm that detects patterns in DNA sequences, protein sequences, or any natural language texts that can be decomposed into letters of a finite alphabet. PTTRNFNDR performs complex mathematical computations, and its processing time increases when input texts become large. To achieve better speed performance, several strategies were applied in the implementation of the program, including parallel operations of binary search trees. A standard binary search tree is not thread-safe due to its dynamic insertions and deletions. Here, we adjusted the standard binary search tree for parallelized operations to achieve improved performance of the PTTRNFNDR algorithm. The method can be applied to other software platforms to quicken data searching through parallel operations of binary search trees when several conditions are met.

Keywords: Computational speed; Data structure; Parallel processing.

Introduction

The binary search tree is one of the most fundamental data structures for dynamic datasets that is used in computer programming. In a binary search tree, every node has a value called a key. The values of all the nodes in a node's left sub tree are less than (or ≤) the node's value, and the values of all the nodes in its right sub tree are greater than the node's value. The binary search tree can support numerical data, character strings, and any other data types for which an order can be defined. It provides an efficient way to store a dataset the size of which is unknown beforehand or changes dynamically. The binary search tree usually provides insertion, searching and deletion as basic operations, which take O(log$n$) in general (O(n) at worst) for each operation. Some forms of binary search trees such as AVL [1] and Red-Black [2] trees can improve searching performance speed by automatically maintaining the balance of the trees as nodes are inserted or deleted.

Parallel computing allows time cost savings. For standard binary trees, however, operations cannot be parallelized directly because race conditions exist when different threads try to insert different children nodes to the same parent node, or when some thread tries to read one node that another thread tries to delete. Researchers have tried to add parallel capability to the binary search trees through changes to the data structure and the corresponding algorithm. For example, the parallel construction of a multidimensional binary search tree was implemented on distributed memory parallel computers to solve applications requiring multidimensional values as keys for the nodes [3]. The overhead of finding the medians of all the nodes could be eased by using "sorting once" or bucket-based strategies. Unfortunately, this solution was designed for distributed memory parallel computers; most current computers are "shared memory" systems. In that light, Solworth and Reagan have performed research to determine the amount of achievable parallelism in operations of a generic tree structure in a shared memory environment [4,5]. To achieve full tree potential, these advances require leveraging arbitrarily large numbers of threads, which can be associated with high computational costs, especially on small processor systems.

A few years ago, we developed PTTRNFNDR, a statistical analysis tool that extracts patterns in biological sequence data files [6]. To ease its calculation burdens, the program uses specially designed data structures and algorithms including a modified binary search tree. The tree allows the program to store fixed length character strings being examined as patterns and account for the number of times they appear in the dataset. The number of different character strings under consideration can be very large. For example, when examining the strings that are only 6 letters long in a genome-derived protein sequence dataset, there could be as many as 64 million different strings that need to be taken into account because each of the 6 positions in the string can be any one of 20 amino acid characters. Thus, the tree can become very large and searching over the tree can consume more computational time as data files grow or as pattern series become more complex.

Implementation of the PTTRNFNDR algorithm required a strategy for shared memory computers having the best performance for storing and searching the string patterns dynamically. Because different operations on the strings occur during different running stages of the program, we were able to design a technique to operate a binary search tree in parallel, which includes parallel operations of insertion, searching, scanning and deletion. Herein, we describe this binary search tree and demonstrate how it improves the speed of the particular biology application. While the technique is applicable to our specific problem, under various conditions the method can be extended to other binary search tree applications with minimal modification.

Materials and Methods

Message Passing Interface (MPI), Open Multi-Processing (OpenMP) and POSIX threads (Pthreads) are the three major parallel programming tools currently available. MPI standard defines a set of library routines to write process-based parallel programs that rely on message-passing to synchronize different processes and transfer data among them. Not depending on shared memory, MPI is most useful in computer clusters instead of single computer servers. By comparison, OpenMP and Pthreads are both threaded, shared-memory programming tools. OpenMP is an API standard design that relies on the compiler to support its preprocessor directives for the threading

***Corresponding author:** Bret Cooper, Soybean Genomics and Improvement Laboratory, USDA-ARS, Beltsville, MD 20705, USA, E-mail: bret.cooper@ars.usda.gov

Received November 03, 2011; **Accepted** November 17, 2011; **Published** November 19, 2011

Citation: Feng J, Naiman DQ, Cooper B (2011) A Parallelized Binary Search Tree. J Inform Tech Soft Engg 1:103. doi:10.4172/2165-7866.1000103

marks added in the normal C or Fortran source files. Pthreads, on the other hand, is a standard on the API libraries provided by the operating systems that implement threading. With today's popularity of multi-core computers, the current trend is to combine MPI with OpenMP or Pthreads to develop high performance applications running on computer clusters. Although OpenMP is easy to code and amendable to task-based applications, we chose Pthreads because parallelization in the data-driven PTTRNFNDR can occur inside the data structure (shown below), and therefore requires fine control on thread synchronization.

In PTTRNFNDR, the first step of the algorithm is to scan the sequence database to find all of the character strings of a prescribed length and their numbers of appearances in the sequence database. To efficiently count the number of appearances for each string, we build a generic binary search tree, the nodes of which are associated with the strings found as the database is scanned. Thus, the data structure of the node consists of a string and the number of occurrences of that string. Alphabetical ordering of the strings is encoded in the tree by requiring that the string associated with any parent node's left child precede the parent's string in alphabetical order, and any parent's string also precede the string associated with its right child. The tree we build is not self-balanced like AVL or Red-Black trees; we allow for the possibility of a parent node with only one child. Still, due to the nature of the protein sequence data we deal with, such nodes are not common.

The tree-building algorithm is initialized with the first string in the database as the root node, with an occurrence count of one. As the database is scanned, each string found is searched for in the current tree. If found, the count at its corresponding node is incremented. If not found, a new node is added in such a way that alphabetic ordering is preserved, and its count is initialized at one. The final tree is obtained once the entire sequence database has been scanned. The algorithm then sweeps through all of the nodes in the second step, and calculates the probability that the each string would appear randomly according to a predetermined probability model. A comparison is made between each string's empirical and expected counts, and a string is chosen as a pattern candidate if it appears significantly more frequently than its probability suggests. When this step is completed, the algorithm will delete the whole tree because all the needed information has been extracted and because the memory occupied needs to be released to generate a new tree for the next sized character strings to be examined. The time costs for insertion, searching, and deleting a binary search tree are usually low. However, the performance of PTTRNFNDR can be improved by parallel processing because the mathematical computing involved in the probability calculations is complex and the datasets are very large.

Our parallelized binary search tree leverages the special features of the PTTRNFNDR algorithm which are clear separations of the tree-building stage, the scanning-computing stage and the tree-deletion stage. Node insertion only happens in the first stage and node deletion only happens in tree-deletion stage while scanning only happens between these two stages. The algorithm for implementing the parallel binary search tree used in PTTRNFNDR consists of the following definition, initialization, insertion, scanning and deletion steps. We also added a searching function below for other applications that might need it when utilizing this parallelized binary search tree structure. Specific details follow.

Data structure definitions and initialization

The parallel operations in PTTRNFNDR are implemented using Linux threads which have little overhead (compared to processes). Mutexes are used to synchronize the threads that might race for the same resource. "Mutex" stands for "Mutual Exclusion" [7]. A mutex is a lock that only one thread may lock at a time. A thread will be blocked (put-on-hold) if it tries to lock a mutex that another thread has already locked. The blocked thread can only continue its execution when the mutex is unlocked. The number of mutexes that can be created is subject to the limit of operating system (OS) and hardware resources. Creating more mutexes usually can reduce the chance of mutex blockings (therefore improving the efficiency of the program), but will also increase the OS cost of managing mutexes. In the published version of PTTRNFNDR, the version we typically use in the laboratory, the number of mutexes that is created is slightly larger than the number of threads. However, for the purpose of this article, in all cases the experimental PTTRNFNDR versions have the same number of mutexes as the number of threads.

All the memory that stores the tree nodes and link information is dynamically allocated using the technology called "obstack" (see http://www.gnu.org/software/hello/manual/libc/Obstacks.html). Each obstack is exclusively used by one thread. The goal is to make the memory allocation in the obstacks thread-safe.

The C code that defines the mutexes, obstack variables, initializations and data structures for tree nodes and linked list nodes is:

```
/*                  obstack section.      */
struct obstack      threadObstack[THREADCOUNT];

/*      type definition section           */
struct  ParTreeST
{
        struct ParTreeST    *left;
        struct ParTreeST    *right;
        char                *key;
        int                 count;
};

struct  ParNodesLinkST
{
        struct ParNodesLinkST       *next;
        struct ParTreeST    *node;
};

typedef struct ParTreeST    ParTree;
typedef struct ParNodesLinkST       ParNodesLink;

/*      static or ordinary global variable definition section.
*/
static  ParTree             *root = (void *)0;
static  ParNodesLink        *head[THREADCOUNT];

static  pthread_mutex_t     mutex[THREADCOUNT];

void    tree_initialize()
{
        int     ii;

        for( ii = 0; ii < THREADCOUNT; ii++ )
        {
                pthread_mutex_init( mutex + ii, (void *)0 );
                head[ii] = (ParNodesLink *)0;
        }
        return;
```

```
}

void    initObstack(int count)
{
        int     ii;

        for( ii = 0; ii < count; ii++ )
        {
                obstack_init( threadObstack + ii );
                obstack_chunk_size( threadObstack + ii ) =
1024*1024;
        }
}
```

Parallel insertion

The insertion operation causes the dynamic set represented by the binary search tree to change. The binary-search-tree property must continue to hold when the tree structure is modified. When the insertions are operated in parallel by several threads, additional care should be taken to guarantee that two or more new nodes will not be attached at the same position of the tree. Otherwise, the conflicts can cause data loss and memory leak. A proper synchronization strategy for the parallel insertions from different threads has to be adopted to prevent any potential conflicts.

A function called TreeParInsert is defined to insert a new node *v* into a binary search tree *T*. The arguments of the function are the address of the node and the ID of the thread that calls the function. The C code of the function TreeParInsert is:

```
void    treeInsertPar(ParTree *v, int len, int threadid)
{
        ParTree  *ptr, *y;
        int             flag;

        for( ptr = root, y = (ParTree *)0;;)
        {
                if( ptr == (void *)0)
                {
                        pthread_mutex_t  *mutexPtr =
mutex +
                        (int)((((unsigned long long)y)/
sizeof(ParTree)) % THREADCOUNT) ;

                        pthread_mutex_lock( mutexPtr );

                        if (y == (void *)0)             /*
empty tree      */
                        {
                                if(root != (void *)0)
/* still empty      */
                                {
                                        root = v;
                                        pthread_mutex_
unlock( mutexPtr );

addLinkedNode(v, threadid);
                                        return;
                                }
                                else
/* not empty anymore */
                                {
                                        pthread_mutex_
unlock( mutexPtr );
                                        ptr = root;
                                }
                        }
                        else if (flag < 0)   /*      left child
*/
                        {
                                if( y->left == (void *)0 ) /*
add a new node here */
                                {
                                        y->left = v;
                                        pthread_mutex_
unlock( mutexPtr );

addLinkedNode(v, threadid);
                                        return;
                                }
                                else
/*      occupied */
                                {
                                        ptr = y->left;
                                        pthread_mutex_
unlock( mutexPtr );
                                }
                        }
                        else                    /* right
child   */
                        {
                                if( y->right == (void *)0 )
/*      add a new node here */
                                {
                                        y->right = v;
                                        pthread_mutex_
unlock( mutexPtr );

addLinkedNode(v, threadid);
                                        return;
                                }
                                else            /*
occupied */
                                {
                                        ptr = y->right;
                                        pthread_mutex_
unlock( mutexPtr );
                                }
                        }
                }

                y = ptr;
                flag = memcmp( v->key, ptr->key, len );
                if( flag == 0 )     /* already in the tree */
```

```
                {
                        /*      need to lock this node
before increasing its count */
                        pthread_mutex_t  *mutexPtr =
mutex +
                        (int)((((unsigned long long)ptr)/
sizeof(ParTree)) % THREADCOUNT) ;

                        pthread_mutex_lock( mutexPtr );
                        ptr->count ++;
                        pthread_mutex_unlock( mutexPtr );
                        return;
                }

                if( flag < 0 )          ptr = ptr->left;
                else                    ptr = ptr->right;
        }
}

void    addLinkedNode(ParTree *v, int threadid)
{
        ParNodesLink    *ptr;

        ptr = (ParNodesLink *) obstack_alloc( threadObstack
+ threadid, sizeof(ParNodesLink) );

        ptr->node = v;
        ptr->next = head[threadid];
        head[threadid] = ptr;
}
```

Like the insertion operation in non-parallel versions of binary search trees, the function TreeParInsert runs in O(h) time on a tree of height h excluding the time for possible mutex-blocking. The chance for a thread to be blocked on a mutex is smaller than $1/n$ (n is the number of mutexes) since threads perform other computations in addition to inserting new nodes.

After the insertion of the new node to the tree, the address of the new node is recorded in a linked list which only takes time O(1). This operation is necessary for the scanning of the whole tree, since otherwise the nodes in the tree have to be scanned through a tree scan algorithm such as DFS (Depth-first Search, such as pre-order, in-order and post-order) or BFS (Breadth-first Search) which cannot be done efficiently in parallel.

Parallel searching

Efficient searching for a given value is the reason to use a binary search tree. This operation does not change any data. Therefore the standard searching procedure can be used directly by the threads. The C code for searching is:

```
ParTree   *treeParSearch(char *key, int len)
{
        ParTree   *ptr;
        int       flag;

        for( ptr = root; ptr != (void *)0; )
        {
                flag = memcmp(key, ptr->key, len);
                if( flag == 0 ) return ptr;
                else if( flag < 0) ptr = ptr->left;
                else ptr = ptr->right;
        }
        return ptr;
}
```

The time cost of a searching operation is O(h)time on a tree of height h.

Parallel scanning

Scanning a whole set of data is a common operation in many applications. This operation on the binary search tree, which is usually performed using a recursive procedure, could have an expensive time cost when the dataset is large or when complex computation is involved to process each node during the scan. For standard binary trees, parallel operation is difficult to implement (if possible at all) due to the tree's special structure. However, since a linked list structure for each thread has already been constructed in the insertion operation, the scan operation can be simply implemented as:

```
void    linkListScanPar(int threadid)

{

        ParNodesLink    *ptr;

        for( ptr = head[threadid]; ptr != (void *)0; ptr = ptr->next )

        printf("Node linked to value [%s] [%d]\n", ptr->node->key,
ptr->node->count);

}
```

The time cost for scanning the whole tree is simply O(n) where n is the number of nodes in the tree.

Tree deletion

In our application the deletion operations are only to delete the whole binary search tree. Therefore, we can use the following O(1) cost method:

```
/*      all the obstacks will be in an "uninitialized" state after the
cleanup.  */

void    cleanUp(int count)

{

        int     ii;

        for( ii = 0; ii < count; ii++ )

        obstack_free( threadObstack + ii, (void *)0 );

}
```

Results

To analyze the performance of the parallelized binary search tree algorithm, we created three versions of PTTRNFNDR, which ran as one thread, two threads, and four threads implemented using the standard multiple thread programming in the GNU C library. There is no other difference among the three versions of PTTRNFNDR. The program was executed in our Dell PowerEdge 2800 server which runs Redhat Enterprise AS 3.0, a 32-bit operating system. There were two dual core CPUs (3.4 GHz) and 6 GB main memory.

The protein sequence databases that we used as input in these experiments were derived from the MSDB database (http://proteomics.leeds.ac.uk/bioinf/msdb.html). The 4 GB memory limit of our 32-

Protein sequences in datafile	Amino acid characters in datafile	1 thread PTTRNFNDR execution time (sec)	2 thread PTTRNFNDR execution time (sec)	4 thread PTTRNFNDR execution time (sec)
48,972	19,106,841	3,101	2,099	1,681
97,809	34,613,670	6,260	4,373	3,322
196,471	66,295,729	11,210	7,800	6,258
295,537	100,137,700	17,645	12,003	9,420
394,898	146,580,372	27,409	19,594	15,589

Table 1: PTTRNFNDR execution time on different protein sequence datafiles. The versions of PTTRNFNDR differed only in the threading of the binary search tree executions.

bit operating system made PTTRNFNDR unable to analyze longer character strings in this database in its full size, so we split the MSDB database into 5 smaller files (Table 1). The smallest and largest files contained approximately 49,000 and 395,000 protein sequences comprising approximately 19,000,000 and 147,000,000 amino acid characters, respectively.

The different sized data files required PTTRNFNDR to utilize different amounts of memory which, in effect, cost time and allowed us to evaluate the cost savings advantage of threading on the parallelized binary search trees (Table 1). The two thread version executed in less time than the one thread version—a 30.7% difference on average. Similarly, the 4 thread version performed faster than the 2 thread version by 22.1%. This clearly shows that the parallel operations of our algorithm on the binary search tree conserved processing time. The Figure shows that the performance was maintained in the 4 thread version even as the size of the input database increased (Figure 1).

Discussion

A fundamental data structure, the binary search tree, was redesigned in this research for possible parallel operations to reduce the time cost of computation in PTTRNFNDR, a statistical pattern detection algorithm. The proposed technique can be regarded as a combination of a binary search tree and several linked lists. The fast searching operation is achieved using the binary search tree structure, fast scanning of the tree is achieved through the linked lists, and the fast deletion of the whole tree is achieved using a special feature of obstack. If the obstack library is not available in an OS, a general memory allocation method (such as malloc) can be used instead. The only difference is that the deletion of the tree will be done by going over all the linked list nodes, which takes O(n) time when the tree has n nodes.

Ideally, speed should increase proportionally with increasing numbers of threads. This, however, is not the case as can be seen in the Table and Figure. There are multiple reasons for this. First, the main memory bandwidth is shared by the 4 processors of the server, which is standard for all shared memory multiple processor computers.

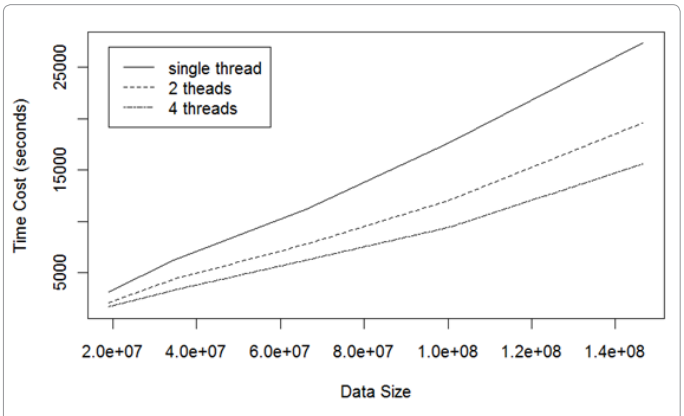

Figure 1: Time cost for executing datafiles of different sizes using different threads for the binary search tree executions in PTTRNFNDR.

Although each processor has its own cache, the low locality nature of the computing in PTTRNFNDR makes the cache ineffective. For example, when it analyzes 6-letter sequences, the binary search tree created in the algorithm has more than 3 million nodes, which means there are more than 3 million unique 6-letter sequences in the searched database. Each node has three pointers: left child, right child and the sequence address. Each node also has a counter to register the number of times each sequence appears in the database. There is also a linked list of nodes, where each node has two pointers, the next node and the corresponding binary search tree node. Together, each unique sequence costs 6 * 4 = 24 bytes in a 32-bit machine. Therefore, the entire data structure requires approximately 80M bytes memory. Yet, the processors cannot store this amount inside their caches. Thus, each insertion or inquiry of the tree causes data transfer from main memory to the processor, and memory bandwidth can easily become the bottleneck of the algorithm. Another reason is the synchronization. When the number of threads increases, the algorithm uses more mutexes to avoid race conditions. The mutexes rely on the services from OS kernel and hardware. These services (system calls) have much higher costs than the normal function calls in the threads. A third cause is that other portions of the program, such as data input, output and word dictionary analysis, were not parallelized. Thus, proving that ideal speed to cost ratios occurred through parallelization would necessitate testing in a perfect computational environment. Since that is beyond the scope of this article, we deem that the results satisfactorily illustrate a relative gain in performance of PTTRNFNDR through the use of a parallelized binary search tree.

The method was successfully implemented in PTTRNFNDR. It most likely can be generalized for other processing applications which build a tree, then search through or scan the whole tree, and finally delete the tree. This unique implementation will allow others to take advantage of multiple processors or multiple core processing capabilities that are common in new computers. This time savings advantage will be meaningful as informational datasets continue to grow in size.

References

1. Adelson-Velskii G, Landis EM (1962) An algorithm for the organization of information. Doklady Akademii Nauk SSSR 146: 263–266.
2. Bayer R (1972) Symmetric binary b-trees: Data structures and maintenance algorithms. Acta Informatica 1: 290-306.
3. Al-Furajh I, Aluru S, Goil S, Ranka S (2000) Parallel construction of multidimensional binary search trees. IEEE Trans Parallel Distrib Syst 11: 136-148.
4. Solworth JA, Reagan BB (1994) Arbitrary order operations on trees. Springer, Berlin / Heidelberg.
5. Solworth JA, Reagan BB (1995) Parallelizing tree algorithms: Overhead vs. Parallelism. Springer, Berlin / Heidelberg.
6. Feng J, Naiman DQ, Cooper B (2007) Probability-based pattern recognition and statistical framework for randomization: Modeling tandem mass spectrum/peptide sequence false match frequencies. Bioinformatics 23: 2210-2217.
7. Mitchell M, Oldham J, Samuel A (2001) Advanced linux programming. New Riders Publishing, Indianapolis, IN.