Parallel algorithms for computing the smallest binary tree size in unit simplex refinement

Ramin Fakhimi

Lehigh University
Department of Industrial and Systems Engineering
Professor: Dr. Roberto Palmieri
December 2020

Abstract

In this project, we considered the problem of finding the smallest binary tree size for the refinement of the unit simplex. At each iteration, the longest edge of the simplex is selected and it is divide to two subsimplices with the longest edge less or equal than a given accuracy. The goal is to find a binary tree which has the minimum size. The size of tree depends on the dimension of simplex and the accuracy and it exponentially as the size of problem increases. Smallest Binary Tree Size (SBTS) problem is an irregular combinatorial optimization problem which requires full enumeration. The characteristics of the SBTSP makes it appealing for parallel computing. In this project, we have implemented three parallel algorithms proposed by Aparicio et al. [2018] using Pthreads and TBB. A recursive parallel algorithm has been tested using TBB where the number of threads is static. The same recursive parallel algorithm also has been implemented using Pthreads with a dynamic number of threads. An iterative version of algorithm also has been implemented using Pthreads. The computational experiments shows iterative version of parallel algorithm outperform other algorithms.

1 Introduction

A simplex is a generalization of the triangle or tetrahedron in higher dimension. Let define the regular n-simplex called S_1 as follows:

$$S_1 = \left\{ x \in \mathbb{R}_+^{n+1} : \sum_{j=1}^{n+1} \frac{\sqrt{2}}{2} \right\} \tag{1}$$

The SBTS problem finds the smallest possible size of a binary tree generated by iterative Longest Edge (LE) refinement of the unit simplex has been introduced in Salmerón et al. [2017].

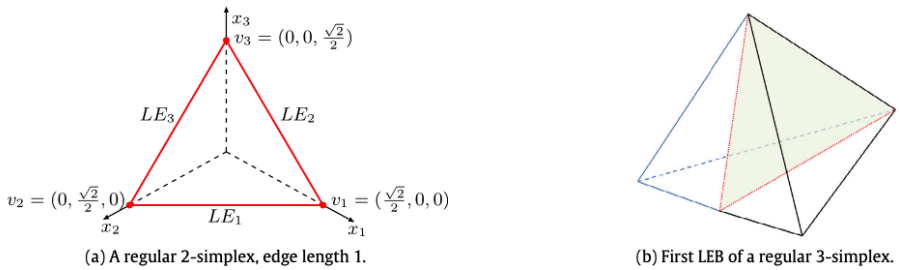


Figure 1: Initial simplices for dimensions three (a) and four (b) [Aparicio et al., 2018].

2 Sequential algorithms for the SBTS problem

For solving the SBTS problem, a tree is built in the forward mode and pruned in the backward mode. The forward mode is used to visit the branches of the tree. In backward mode the tree is traversed from leaves to root, to track the smallest branch length. The following BnB steps are followed to achieve this:

Division: In forward mode, every longest edge (LE) option of a simplex is checked. The bisection of each LE of the current simplex generates a pair of sub-simplices or children.

Bounding: In backward mode, lower and upper bounds on the size of the binary sub-trees of a node can be calculated when all its sub-trees (generated by different LEB) have been built.

Selection: In forward mode, the deepest simplex or node of the general tree is selected for division using a Depth-First search. Depth-First search has less memory requirement than other approaches [Ibaraki, 1976].

Rejection: In backward mode, only the lower bound of a node of the general tree is considered in the calculation of the SBTS problem solution. All these sub-trees can be removed from the search, as they have already completely been evaluated.

Termination: In forward mode, a simplex smaller than a given precision ϵ is not further divided. In backward mode, the algorithm finishes when the bounding rule is applied to the root node. Then, its lower bound plus one is the solution of the SBTS problem.

Figure 2 illustrates the smallest binary tree on a 2-simplex with $\epsilon = 2$.

2.1 Recursive smallest binary tree size algorithm

The algorithm starts by checking if the LE length of the current simplex S_i is less than ϵ. In such case, it returns 1 as the length of its tree. Otherwise,

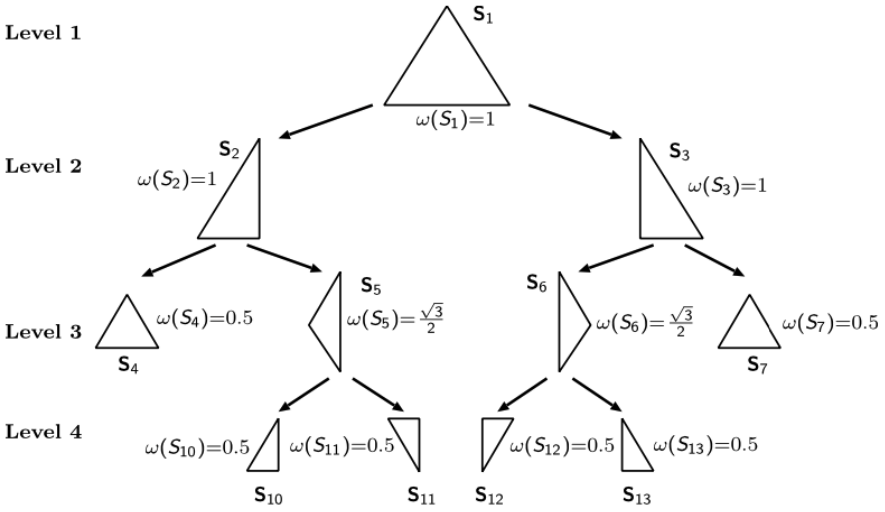


Figure 2: Binary tree generated by the SR-LEB algorithm on a 2-simplex with $\epsilon = 0.5$ [Aparicio et al., 2018].

S_i is divided by each of its longest edges longest edges, generating two sibling instances of a binary tree. The size of the sub-tree of each sibling is determined by a recursive call, which automatically leads to a Depth-First traversal of the tree. The sum of sibling sub-tree sizes is stored. Finally, the smallest value of sub-tree size over the bisected edge is returned, for considering the current node 1 adds to the result.

Algorithm 1 RecSeqSBTS (S_i, ϵ)

Require: (S_i, ϵ)
1: **if** $\omega(S_i) \leq \epsilon$ **then**
2: **return** 1
3: **for** each LE_h of S_i **do**
4: $\{S_{2i}, S_{2i+1}\} \coloneqq$ Bisect$(S_i,\ LE_h)$
5: $r_{2i,h} \coloneqq$ RecSeqSBTS (S_{2i}, ϵ)
6: $r_{2i+1,h} \coloneqq$ RecSeqSBTS (S_{2i+1}, ϵ)
7: **return** $1 + \min_h r_h$

2.2 Iterative smallest binary tree size algorithm

The iterative sequential algorithm follows the described BnB approach. It uses dynamic memory to store the connections among nodes of the general tree, instead of using the procedure calls to determine those connections. Now, additional links to the parent node appear, facilitating the traversal of the tree

when looking for pending work is performed in parallel.

Algorithm 2 CreateNode (S_i, ParentNodePtr, ChildIndInParent)

Require: (S_i, ParentNodePtr, ChildIndInParent)
1: Create new node N with S_i
2: set parent node pointer to ParentNodePtr
3: IsChildrenCreated := **false**
4: SBTS := 0
5: longest edges are calculated
6: **return** N

Figure 3 demonstrate a node data structure for the case of a simplex with three longest edges. Algorithm 2 creates a node with the new simplex, a link to its parent node and its position in the ChildIndInParent vector of the parent node. The size of the sub-tree rooted at this node (SBTS) is set to 0, because it is unknown and IsChildrenCreated is set to false, because its child nodes were not created. Algorithm 3 creates the children nodes bisecting the simplex by each of its longest edges. At the end, the simplex is removed in order to free memory.

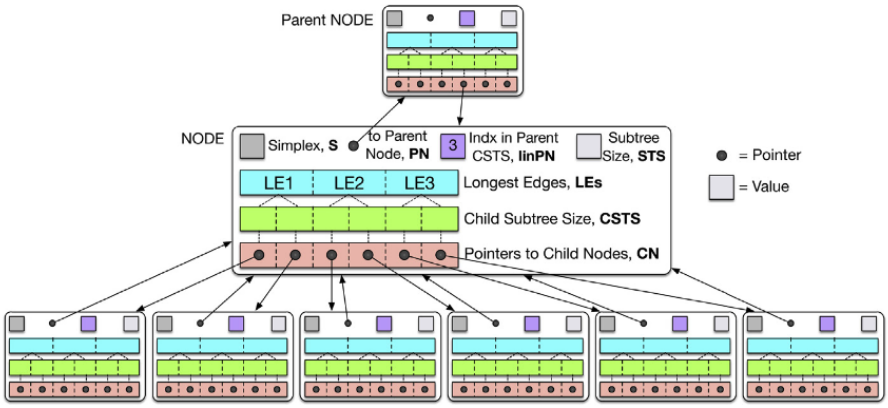


Figure 3: Node Data Structure [Aparicio et al., 2018].

Algorithm 3 CreateChildNodes (N)

Require: (N)
1: **if** $\omega(S_i) > \epsilon$ **then**
2: **for** edge LE_h of S_i **do**
3: $\{S_{2i}, S_{2i+1}\} \coloneqq$ Bisect(S_i, LE_h)
4: $N_{2i} \coloneqq$ CreateNode(S_{2i}, N_{ptr}, h)
5: IsTwin $\coloneqq$ **true**
6: **if** bisection generates different simplices **then**
7: $N_{2i+1} \coloneqq$ CreateNode(S_{2i+1}, N_{ptr}, h)
8: IsTwin $\coloneqq$ **false**
9: Add generated node(s) to the vector of ChildernPtrToNode
10: Add a tuple of (0, 0, IsTwin) to the vector of ChildrenSBTS
11: **if** All the edges of S_i are equal **then**
12: **break**
13: **else**
14: SBTS $\coloneqq$ 1
15: **delete** S_i and remove vector of longest edges
16: IsChildrenCreated $\coloneqq$ **true**

The iterative algorithm to calculate the smallest binary tree size is presented in Algorithm 4. The algorithm starts with creating a new node. Then it moves to an infinite loop where every time first it check whether the children of the current node is created and it creates them if not. At the next step, it picks one of the unseen children. If there is no unseen children then it calculates the SBTS of the current node otherwise it does a depth search and goes for a child of node's children. If the current node is the initial node then the algorithm breaks the loop and return the SBTS, otherwise it moves backward and goes to the parent of the current node and update its vector of ChildrenSBTS. To avoid memory leak, it release the memory of the current node. In the implemented algorithm, we reduced the computation by considering the following improvements:

- It is enough to consider one of the edges of a simplex that all of its edges have the same length.
- If two siblings have the same shape, then it is enough to consider one of them

Algorithm 4 IterSeqSBTS (S_1, ϵ)

Require: (S_1, ϵ)
1: $N \coloneqq$ CreateNode (S_1, ParentNodePtr, ChildIndInParent)
2: **while true do**
3: **if** the children of node N is not created **then**
4: CreateChildNodes (N)
5: SBTSCalulationStep := **true**
6: **for** child in the vector of ChildrenSBTS **do**
7: Set N to the pointer of a child that its SBTS is not calculated
8: SBTSCalulationStep := **false**
9: **break**
10: **if** SBTSCalulationStep is **true then**
11: **continue**
12: **else if** SBTS is not 1 **then**
13: Calculate the SBTS of the Node N
14: **if** N is not the initial node **then**
15: Update the vector of ChildrenSBTS of the parent node of N
16: **delete** node N
17: Set N to its parent node
18: **else**
19: **break**
20: **return** SBTS

3 Parallel algorithms for the SBTS problem

Parallel computing can be applied to solve the SBTSP, because several branches of the general tree can be processed in parallel. The SBTS problem is hard to solve because of differences in branches length and it requires dynamic load balancing [Crainic et al., 2006]. There are several libraries and programming models for shared memory architectures. However, the authors tried TBB and Pthreads. TBB can be considered task-based frameworks, in which the user can select the number of threads, but express parallelism in terms of tasks that are executed by these worker-threads. The authors used TBB since user-level tasks are more lightweight than the kernel-level worker-threads which are usually just dispatched once, and the worker-threads can implement automatic task balancing via work-stealing. They also chose Pthreads, because they have found that TBB also have some shortcomings. TBB requires the correct selection of a Cut-off. A small Cut-off could restrict parallelism whereas a large one might saturate the system with an excessive number of fine grained tasks. Second, TBB's work-stealing algorithm can perform sub-optimally for large enough trees. Summarizing, in TBB, an idle worker-thread steals the shallower sub-tree from the pending queue of another worker. From a memory footprint standpoint, this is not a good idea and it may cause out-of-memory problems.

3.1 Recursive algorithm using TBB

In this algorithm, the number of threads is static and is set by user. The parallel programming approach in TBB is mostly based on tasks rather than threads. TBB library provides both the high-level and low-level programming approach. The way that is implemented in this project is the high-level version of TBB. At each iteration of the algorithm, TBB creates two tasks corresponding to the bisection of current nodes. The only difference of the parallel version of the algorithm and its sequential version is that sub-tree computations are conducted in new spawned tasks. A cut-ff parameter is also defined to improve the performance of the algorithm. Whenever algorithm reaches at some precision then it shifts to sequential version. This approach limits the tree depth in parallel algorithm.

Algorithm 5 RecParTBBSBTS (S_i, ϵ)

Require: (S_i, ϵ)
1: **if** $\omega(S_i) \leq \epsilon$ **then**
2: **return** 1
3: **else if** $\omega(S_i) \leq$ CutOff **then**
4: RecSeqSBTS (S_i, ϵ)
5: **for** each LE_h of S_i **do**
6: $\{S_{2i}, S_{2i+1}\} \coloneqq$ Bisect$(S_i,\ LE_h)$
7: $r_{2i,h} \coloneqq$ spawn(createNewTask RecParTBBSBTS (S_{2i}, ϵ))
8: $r_{2i+1,h} \coloneqq$ spawn(createNewTask RecParTBBSBTS (S_{2i+1}, ϵ))
9: Wait until the spawned tasks are done.
10: **return** $1 + \min_h r_h$

3.2 Recursive algorithm using Pthreads

Algorithm 7 illustrates the recursive parallel algorithm similar to Algorithm 5. Each thread runs the recursive sequential algorithm with the option to create new threads to process pending work. As CreatedThreadNum variable is shared by all threads, the first thread will be the one in charge of creating a new thread. In this way, dynamic load balancing is inherent to dynamic thread creation. A thread decrements the value of CreatedThreadNum when it finishes its work, returns its result and dies. A thread performs a recursive call when it cannot create a new thread or it is processing the last LE bisection of a simplex. In this way, a task recycles itself as the last child task, instead of spawning it. Therefore, the last child is immediately executed in the same thread/core. One of the caveats of this algorithm is that it cannot move backward until it obtains the results of its children. The difference with the TBB version is that the Pthread implementation performs Breadth-First search at deeper levels of the general tree, hence requiring less memory.

Algorithm 6 RecParPthSBTS (S_i, ϵ)

Require: (S_i, ϵ)
1: **if** $\omega(S_i) \leq \epsilon$ **then**
2: **return** 1
3: **else if** $\omega(S_i) \leq$ CutOff **then**
4: RecSeqSBTS (S_i, ϵ)
5: **for** each LE_h of S_i **do**
6: $\{S_{2i}, S_{2i+1}\} \coloneqq$ Bisect(S_i, LE_h)
7: **if** CreatedThreadNum $<$ ThreadNum **then**
8: CreatedThreadNum++
9: $r_{2i,h} \coloneqq$ CreateThread(RecParPthSBTS (S_{2i}, ϵ))
10: **else**
11: $r_{2i,h} \coloneqq$ RecParPthSBTS (S_{2i}, ϵ)
12: **if** CreatedThreadNum $<$ ThreadNum **and** LE_h is not the last one **then**
13: CreatedThreadNum++
14: $r_{2i+1,h} \coloneqq$ CreateThread(RecParPthSBTS (S_{2i+1}, ϵ))
15: **else**
16: $r_{2i+1,h} \coloneqq$ RecParPthSBTS (S_{2i+1}, ϵ)
17: Wait for the created threads.
18: CreatedThreadNum$--$
19: **return** $1 + \min_h r_h$

3.3 Iterative algorithm using Pthreads

This algorithm is a parallel extension of iterative sequential Algorithm 4. The main goal is to avoid drawbacks of Algorithm 7. To achieve this goal, a thread should not wait for the results from the threads it created. Therefore, the thread returning the result of a node is the one noticing that the work of a node has already been done. That thread can be different from the thread which created the node. Additionally, threads are eager to look for work by traversing the general tree.

Instead of using a recursive implementation, the algorithm is iterative and its dynamic memory allocation facilitates threads to traverse the tree, because a thread can move to sub-trees different from the one where it was created. The general idea is to move to the parent node looking for work when there is no work in the current node. It means threads looking for work visit the general tree from deeper to higher levels. A thread reaching the root node without having found work dies and a new thread can be created. This strategy, that we call hybrid search, effectively applies a Depth-First search with the possibility of a Breadth-First search at the deepest nodes ready to be processed.

Algorithm 7 nextChildToProcess (NodePtr)

```
Require: (NodePtr)
 1: if TryLock to access NodePtr is not successful then
 2:     return -1
 3: else
 4:     IsDone := true
 5:     for Child in Children of NodePtr do
 6:         if ChildSBTS is equal to -1 then
 7:             IsDone := false
 8:         else if ChildSBTS is equal to 0 then
 9:             ChildSBTS is equal to -1
10:             return -1
11: return -2
```

4 Numerical Results

We have implemented all the discussed algorithms. For doing the computational experiments, we used Sunlab servers. Table 1 contains the summary of results for four different SBTS problems. Surprisingly, in three SBTS problems the recursive algorithms are different from their iterative counterpart. However, the difference is not big. We investigated this issue by changing some parameters of the algorithms. We could not get a definitive answer for it. However, some parameters and subroutine like finding symmetric sub-simplices affected the results.

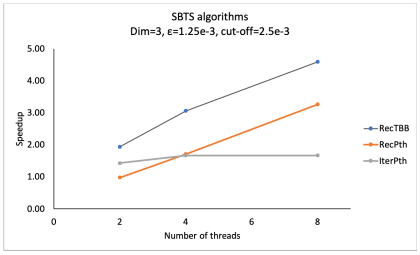
As we expected, the IterPth algorithm outperforms recursive parallel algorithms. As shown in Figure 4, IterPth algorithm also achieved speedup over its sequential version. However, recursive versions of algorithms could not gain speedup in all the cases. One can realize that even a high-level version of TBB can outperform Pthreads parallel programming library. In the paper, the authors also compared different memory management libraries. However, we did not do those experiments in the interest of time.

Algorithm 8 IterParPthSBTS (NodePtr)

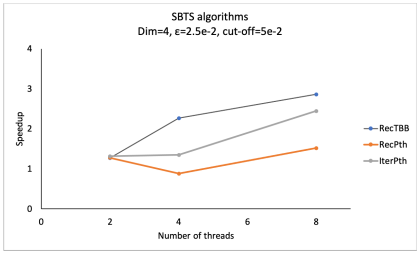
```
Require: (NodePtr)
 1: while true do
 2:     if the children of node NodePtr is not created then
 3:         CreateChildNodes (NodePtr)
 4:     nextChild := nextChildToProcess(NodePtr)
 5:     while nextChild ≥ 0 and CreatedThreadNum < ThreadNum do
 6:         CreatedThreadNum++
 7:         Set NodePtr to the corresponding Child of NodePtr
 8:         CreateThread (IterParPthSBTS (NodePtr) )
 9:         nextChild := nextChildToProcess(NodePtr)
10:     if nextChild is equal to -1 then
11:         if NodePtr is the initial node then
12:             CreatedThreadNum−−
13:             break
14:     else if nextChild ≥ 0 then
15:         Set nextChild to one of its children
16:         continue
17:     else
18:         if IsDone is false then
19:             CreatedThreadNum−−
20:             break
21:         else
22:             Set NodePtr to its parent node
23:         Calculate SBTS of NodePtr
24:         if NodePtr is not correspond to the inital node then
25:             Update the SBTSChildren vector of its parent node
26:             delete node NodePtr
27:             Set NodePtr to its parent node
28:         else
29:             break
```

5 Conclusion

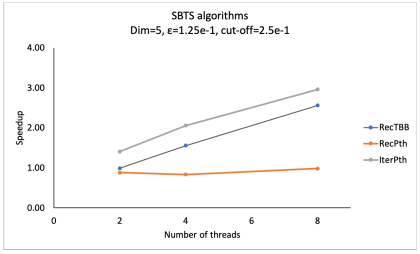
The smallest binary tree size problem is an NP-hard problem that the its feasible space exponentially grows as the size of problem increases and accuracy decreases. In this project, we have implemented three parallel algorithm using TBB and Pthreads libraries. The sequential versions of proposed algorithms seems very straightforward while their parallel versions have lots of details. One needs to consider several safeguards to avoid crashing the problems.



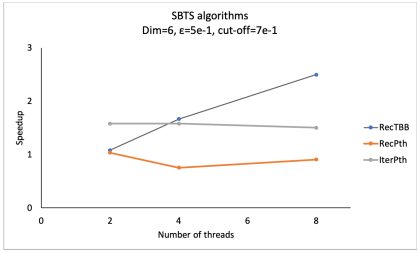
(a) Dim=3, ϵ=1.25e-3, cut-off=2.5e-3



(b) Dim=4, ϵ=2.5e-2, cut-off=5e-2



(c) Dim=5, ϵ=1.25e-1, cut-off=2.5e-1



(d) Dim=6, ϵ=5e-1, cut-off=7e-1

Figure 4: Obtained speedup in the parallel versions of algorithms.

Problem			Algorithm	SBTS	Run time w.r.t. # of threads (s)			
Dim	ϵ	cut-off			sequential	2	4	8
3	1.25e-3	2.5e-3	IterPth	3145727	10	7	6	6
			RecPth	3145727	101	103	59	31
			RecTBB	3145727	101	52	33	22
4	2.5e-2	5e-2	IterPth	888703	105	80	78	43
			RecPth	887975	1127	885	1279	743
			RecTBB	887975	1127	886	498	394
5	1.25e-1	2.5e-1	IterPth	89375	1278	908	621	432
			RecPth	88859	16730	190063	20073	17026
			RecTBB	88859	16730	16898	10763	6538
6	5e-1	7e-1	IterPth	1199	30	19	19	20
			RecPth	1183	449	435	598	496
			RecTBB	1183	449	416	270	180

Table 1: Run time and solution of the SBTS problem using the proposed algorithms

References

Guillermo Aparicio, Jose MG Salmerón, Leocadio G Casado, Rafael Asenjo, and Eligius MT Hendrix. Parallel algorithms for computing the smallest binary tree size in unit simplex refinement. *Journal of Parallel and Distributed Computing*, 112:166–178, 2018.

Jose MG Salmerón, Guillermo Aparicio, Leocadio G Casado, Inmaculada García, Eligius MT Hendrix, et al. Generating a smallest binary tree by proper selection of the longest edges to bisect in a unit simplex refinement. *Journal of Combinatorial Optimization*, 33(2):389–402, 2017.

Toshihide Ibaraki. Theoretical comparisons of search strategies in branch-and-bound algorithms. *International Journal of Computer & Information Sciences*, 5(4):315–344, 1976.

Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel branch-and-bound algorithms. *Parallel combinatorial optimization*, 1:1–28, 2006.