An Architecture Independent Study of Parallel Segment Trees

Alexandros V. Gerbessiotis

Computer and Information Science Department

New Jersey Institute of Technology

Newark, NJ 07102.

May 24, 2002

Abstract

In this work we study the computation, communication and synchronization requirements related to the construction and search of parallel segment trees in an architecture independent way. Our proposed algorithms are optimal in space and time compared to the sequential algorithms utilized to solve the corresponding problems and are described in the context of the Bulk-Synchronous Parallel (BSP) model of computation. Our methods are more scalable (i.e. work for larger values of p relative to n, the problem size) than other segment tree related algorithms that have been described on other realistic distributed-memory parallel models and provide a natural way to approach searching problems on the BSP model that maintain a balanced query load among the processors.

Keywords: Architecture independent parallel algorithms, parallel algorithm design, parallel segment trees

1 Introduction

Architecture dependent realizations of distributed data structures have been addressed in the past for particular network architectures, i.e., hypercubes and meshes [1, 6]. Architecture dependent features involved in these implementations, make the migration of these results to realistic parallel computer models and machines non-trivial. On the Bulk-Synchronous Parallel (BSP) model the problem of multi-way search has been examined in the context of d-ary tree structures and optimal results have also been obtained in [2] . In [9, 14] multi-way search has been studied on a wider class of graphs, called ordered h-level graphs [6], and optimal results have been derived. In this work we extend the problem of parallel searching under realistic parallel computer models [2, 9, 14] to work on parallelized geometric data structures, namely parallel segment trees. In particular, we propose efficient parallel algorithms for some well known and studied sequential problems related to the construction and search of segment trees [19, 3].

We thus study the computation, communication and synchronization requirements related to the construction and search of parallel segment trees on the BSP model. A parallel segment tree of size n is a sequential segment tree [19] that is evenly stored among the p processors of a parallel computer [7]. Queries searching such a tree are presented in a group of size m that is evenly distributed among the p processors. Depending on the particular problem that is to be solved with a segment tree, the structured information associated with each of its nodes may vary. It can be a scalar value or a list of geometric structures (i.e. segments) in sorted or no particular order.

We are proposing solutions to problems that will utilize different information structures of a segment tree by building a parallel segment tree with these on a set of n non-intersecting line segments in the plane and a direction D. For easiness of presentation, in the remainder, we shall assume, without loss of generality, that D is the y axis. The parallel segment tree, along with its information structure associated with its nodes, will be evenly distributed among the p processors of the parallel machine. Subsequently, for a set of m query points, evenly distributed among the processors, the segment tree with its information structure will be accessed to solve one of the following problems. Other interesting applications of segment trees can be found in [19].

- *the Counting problem* : Count the number of input segments a ray starting from each query point intersects in direction D.
- *the First-hit problem* : Report the first segment a ray starting from each query point

intersects in direction D.

Segment trees on coarse-grained parallel models have been studied in the past. In [7] they are used for the solution of problems related to trapezoidal decomposition and the bichromatic line segment intersection of a set of non-intersecting line segments in the plane. On a coarsed-grained multicomputer (CGM) on p processors, the trapezoidal decomposition of n non-intersecting line segments in the plane is obtained in time $O(n \log n \log p/p) + T(n \log p, p)$, for $p^2 < n$, where $T(n, p)$ is the time required to sort n keys on p processors with n/p keys assigned to each processor. A segment tree is built to obtain the desired time bound. This algorithm, as noted in [7], naturally extends to solve the First-hit problem within the same time bound for $m = n$. In [5] an algorithm for the next element search problem (First-hit, in the terminology adapted in this work) is presented. On a p-processor CGM n queries are satisfied in $O(1)$ communication rounds and space $O(n \log n/p)$. The algorithm first builds a segment tree using a construction based on ideas of [7]; the segment tree construction algorithm utilizes a global sort operation similarly to [7] which requires time equal to $T(n \log p, p)$. Under the assumption that segment coordinates are such that count-sort/radix-sort algorithms can be used for sorting segments/points rather than generalized sorting algorithms (such as heapsort, mergesort) the bound for $T(n \log p, p)$ used in [5, 4] is that of using radix-sort to sort $n \lg p$ keys on p processors. The described algorithm in [5, 4] also works for processor ranges such that $p^2 < n$ or equivalently $p = O(\sqrt{n})$.

Our methods in this work for solving segment-tree related problems extend past results on searching static data structures on the BSP model [9]. They are also more scalable than other parallel segment tree algorithms such as those of [5, 4, 7]. The processor range that our methods are applicable is much higher than $p = O(\sqrt{n})$, namely $p = O(n/\lg n)$. In addition we make no special assumptions about the segment coordinates like those made in [5, 4]. Our algorithms are also optimal compared to the corresponding sequential algorithms. In the BSP terminology, we claim that our algorithms for the Counting problem are one-optimal for a wide range of the BSP parameters p, L and g. The BSP algorithm for segment tree construction for the First-hit problem is by a $\log n$ factor slower than the best sequential algorithm but one-optimal compared to a practical sequential algorithm, as are the results in [7] and also in[5, 4] if the assumption on the segment coordinates in the later work is relaxed. The only known to the author PRAM algorithm for the bichromatic line segment intersection problem (that belongs to a class of problems whose solution uses segment trees) works on a CREW-PRAM and solves the counting problem in $O(\log n)$

time with $O(n + k/\log n)$ processors, where k is the number of intersections.

Our methods also suggest a natural way to approach geometric searching on the BSP model that maintains a balanced load among the processors not only for storing the data structure to be searched but also the queries that access it. Extensions of our methods to solving other geometric data structure problems, such as those reported in [7] (range query reporting, bichromatic line segment intersection problems) seems quite tractable and immediate. Our architecture independent algorithms can be made to work on other similar practical parallel models including shared-memory ones and offer us valuable insights on designing parallel algorithms and data-structures that take into consideration both communication and synchronization issues.

1.1 The BSP Model

The BSP model of computation has been proposed [20, 21] as a unified framework for the design, analysis and programming of general purpose parallel computers. It allows the design of algorithms that are both scalable and portable. In a BSP program, processors jointly advance through its various phases called *supersteps* with the required computations and remote communication occurring between them; at the end of a superstep processors synchronize themselves. A BSP computer consists of *p processor/memory components*, each one corresponding to a sequential computer, *a communication network with a router* that delivers messages point-to-point between the components, and facilities for *global synchronization* of all or a subset of the components.

The tuple (p, L, g) characterizes the behavior and performance of a BSP computer. Parameter L is the minimum time between successive synchronization operations, and g is the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered. A lower bound on the value of L is the time for a remote operation to become effective and is thus dependent on the properties of the interconnection network. The time for synchronization also poses a lower bound on the effective value of L. The value of g is measured while the network is in a steady state, i.e. latency contributions become insignificant and parameter L is made large enough for the theoretical bound on g to be realizable. The theoretical definition of g relates to the routing of h-relations; when each processor sends or receives at most h messages an h-relation is realized and the cost assigned to this communication is gh provided that $h \geq h_0$, where h_0 is a machine dependent parameter. Otherwise the cost of communication is L. For any BSP computer, the values of L and g are likely to be non-decreasing functions of p.

A cost of $\max\{L, x + gh\}$ basic time units is assigned to a superstep S, where x is the maximum number of basic computational operations executed by any processor during S, and h is the maximum number of messages sent or received by any processor. If h_s is the maximum number of messages sent and h_r is the maximum number of messages received by any processor during S, then $h = \max\{h_s, h_r\}$. The maximum size of a superstep depends on the problem in hand. Under the BSP model the objective is to maximize the size of supersteps, decrease their number and increase processor utilization. The description of BSP algorithms can be simplified by separating computation and communication [8] and thus assuming that each superstep contains either local computations or communication.

1.2 Optimality and Analysis of BSP Algorithms

The performance of a BSP algorithm P is described in three parts. An algorithm S (say, a sequential one) with which we are comparing P is first specified. The model of computation used for both algorithms is then defined and the basic computational operations that will be counted in both P and S are also described and the charging policy is made explicit. Second, two ratios π and μ are specified. The former, π, is the ratio between the computation time C_P, of the BSP algorithm, over the time C_S of the comparing sequential algorithm divided by p, i.e., $\pi = pC_P/C_S$. The latter, μ, is the ratio between the communication time M_P required by the communication supersteps of the BSP algorithm and the computation time of S divided by p, i.e., $\mu = pM_P/C_S$. When communication time is described, it is necessary that the amount of information that can be transmitted in a single message be made explicit. Finally, conditions on n, p, L and g are specified that are sufficient for the algorithm to be plausible and the claimed bounds on π and μ to be valid. Corollaries describe sufficient conditions for the most interesting optimality criteria, such as c-optimality, i.e., $\pi = c + o(1)$ and $\mu = o(1)$. All asymptotic bounds refer to problem size n, as $n \to \infty$. As performance ratios between a parallel and a sequential algorithm are expressed in terms of π and μ, operations may be defined in a higher level of abstraction than machine level instructions.

1.3 Primitive Operations

Throughout this work, lg and log denote logarithms to the base two and e respectively. A dominant term in an integer sorting BSP algorithm to be claimed is contributed by sequential count-sort performed independently at each processor; such an algorithm is to be referred as COUNT_SORT [9]. $O(n)$ time units are charged for stable sorting n integers in the range $[0, 1, \ldots, n-1]$ by any sequential count-sort algorithm [17]. Radix-sort techniques are used to sort n integers in the range $[0, \ldots, n^{\alpha}-1]$, for some positive constant α, within $\lceil \lg \lceil n^{\alpha} \rceil / \lg n \rceil$ iterations of COUNT_SORT; such an algorithm is to be referred as RADIX_SORT. A sequential sub-quadratic time worst-case sorting algorithm (say heapsort, merge-sort) [17], to be called SUB_QUADRATIC_SORT, that can sort $\lceil \sqrt{n} \rceil$ keys in $o(n)$ time units is also utilized [9]. For other operations, $O(1)$ time units will be charged for operations over an associative operator and one time unit for each comparison. In this work a number of BSP algorithms for several basic operations like *broadcasting*, *parallel-prefix* and *integer-sorting* (*parallel radix-sort*) will be used. A fully detailed account of such operations along with additional results can be found in [11, 16].

(1) A p-processor BSP algorithm for broadcasting an n-word vector, requires communication time at most $M^n_{brd}(p) = 2 \max\{L, 2gn\} \cdot [n > 1] + h \max\{L, 2gt\} \cdot [n < p]$ for any $t \le p$, where $h = \lceil \log_t ((t-1)\lceil p/n \rceil + 1) \rceil - 1$, and the bracketed notation [S] evaluates to 1 if S is true, and 0 otherwise. The overall time complexity of the algorithm is denoted by $T^n_{brd}(p) = M^n_{brd}(p)$. For $n = 1$, we write $T_{brd}(p)$.

(2) A p-processor BSP algorithm for realizing n disjoint parallel-prefix operations, each of size p, over any associative operator whose application to two arguments take $O(1)$ time, requires time at most $C^n_{ppf}(p) = 2 \max\{L, O(n)\} \cdot [n > 1] + 2h \max\{L, O(t)\} \cdot [n < p]$ for computation and $M^n_{ppf}(p) = 2 \max\{L, 2gn\} \cdot [n > 1] + 2h \max\{L, 2gt\} \cdot [n < p]$, for communication for any $t \le p$, where $h = \lceil \log_t \lceil p/n \rceil \rceil$. The overall time complexity of the algorithm is denoted by $T^n_{ppf}(p) = C^n_{ppf}(p) + M^n_{ppf}(p)$. For $n = 1$, we write $T_{ppf}(p)$.

(3) A p-processor BSP stable radix-sort based algorithm sorts n integers in the range $[0, 1, \ldots, R-1]$ in computation time equal to $r\left(3 \max\{L, O\left(\max\{n/p, t \log_t p\}\right)\} + C_{ppf}(p)\right)$ and communication time $r\left(2 \max\{L, g \ \max\{n/p, t \log_t p\}\} + \max\{L, gn/p\} + M_{ppf}(p)\right)$, for any t such that $2 \le t \le p$, where $r = \lceil \lg R / \lg \max\{(\lceil n/p \rceil, t \lceil \log_t p \rceil)\} \rceil$.

A general parallel sorting algorithm that can sort any data type will also be utilized. We can either use the one-optimal randomized algorithm in [8] or the one-optimal deterministic ones in [10, 12] or the fully scalable but c-optimal one in [15]. The time required for a BSP sorting algorithm to sort n evenly distributed keys on p processors is denoted by $T(n,p)$.

2 Segment Trees

We briefly discuss sequential segment trees [19, 3] that were introduced for the purpose of solving problems related to intervals on the real line. Let $x(q)$, $y(q)$ be the x and y coordinates of a query point q. Let δ be a line segment, and let $\mathit{left}(\delta)$ and $\mathit{right}(\delta)$ be its two end-points. Then $x(\mathit{left}(\delta))$ is the x coordinate of its left end-point. Let $x(\delta)$ denote the interval $[x(\mathit{left}(\delta)), x(\mathit{right}(\delta))]$. Let T be a balanced binary tree and let v be one of its internal nodes or its root. We denote by $l(v)$ and $r(v)$ the left and right children of v, and $\mathit{root}(T)$ its root. The level of $\mathit{root}(T)$ is zero; *a numbering* of the nodes of T would start from level zero and proceed level by level left to right.

Let I be a set of r non-intersecting line segments in the plane. Let $x_1, \ldots, x_t$, $t \leq 2r$, be the distinct x coordinates ([3] describes ways to handle the duplicates) of the $2r$ end-points of the segments. Elementary intervals are then formed [3]: $(-\infty, x_1), [x_1, x_1], (x_1, x_2), \ldots, (x_t, +\infty)$. The number of elementary intervals is $2t+1$, i.e. at most $4r+1$ such intervals are constructed from the original set of r line segments. Subsequently, let the number of the formed intervals be n.

A segment tree $S(T)$ consists of a base tree BT and the structures associated with the nodes of BT. BT is a balanced binary tree with n leaves, each leaf corresponding to one elementary interval with the leftmost leaf corresponding to the leftmost interval. Let the elementary interval of leaf v be denoted by $\mathit{Int}(v)$. The interval $\mathit{Int}(u)$ of an internal node u is the union of all elementary intervals of the leaves in the subtree of BT rooted at u. Each internal node or leaf of BT, maintains pointers to its left and right child, stores $\mathit{Int}(u)$ and may also store additional information denoted by $I(u)$ and $c(u)$. Both or one of $c(u)$ and $I(u)$ may be stored at u. $I(u)$ denotes the list of line segments stored at u and $c(u)$ the cardinality of $I(u)$. Other application dependent information may also be stored at u. Hereafter, we shall use the term *base tree* BT of $S(T)$ to include the structure of BT and the $\mathit{Int}(u)$ intervals but neither $I(u)$ nor $c(u)$. We shall collectively refer to the latter information as the *auxiliary structure* of $S(T)$.

A sequential algorithm for constructing BT given n elementary intervals is straightforward and requires time and space $O(n)$. In order to construct $S(T)$ the line segments need to be inserted in BT. For each line segment δ a call to BUILD_STREE ($\mathit{root}(S(T))$,δ) is issued. In step 2 of

BUILD_STREE the auxiliary structure (eg. $I(v)$ or $c(v)$) is updated. Segment tree properties [3, 19] show that line 2 of BUILD_STREE will be executed $O(n \lg n)$ times when a segment tree on n intervals is being built, with each δ executing at most twice line 2 of BUILD_STREE and visiting at most four nodes in each level. A node u at level i will be visited by $O(n/2^i)$ segments.

```
beginBUILD_STREE (v,δ)
/* v is a node of segment tree S(T) being built and δ a line segment */
1.    if  Int(v) ⊆ x(δ)
2.        then  Update auxiliary structure of v with δ;
3.        else  begin
4.                  if  Int(l(v)) ∩ x(δ) ≠ ∅
5.                      BUILD_STREE (l(v),δ);
6.                  if  Int(r(v)) ∩ x(δ) ≠ ∅
7.                      BUILD_STREE (r(v),δ);
8.              end
end BUILD_STREE
```

Figure 1: Procedure BUILD_STREE.

For a query point q a call to QUERY_STREE ($root(S(T))$,q) is issued. Query q visits $O(\lg n)$ nodes in the tree.

```
beginQUERY_STREE (v,q)
/* v is a node of segment tree S(T) being built and q a query point */
    Report as appropriate auxiliary structure of v;
    if  (v internal) ∧ x(q) ∈ Int(l(v))
        then      QUERY_STREE (l(v),q);
        else      QUERY_STREE (r(v),q);
end QUERY_STREE
```

Figure 2: Procedure QUERY_STREE.

3 Parallel Segment Trees

The first step in building a parallel segment tree on a p-processor BSP computer involves the construction of base tree BT. More formally, r line segments are uniformly but otherwise arbitrarily distributed among the processors so that each processor stores at most $\lceil r/p \rceil$ segments whose endpoints are then extracted in time $O(r/p)$ and sorted in time $T(2r, p)$. Subsequently, $r \leq n \leq 4r + 1$ elementary intervals are formed [3] and stored evenly among the p processors so that each processor holds at most $\lceil n/p \rceil$. We assume, without loss of generality, that n is a power of two; if this not the case, we may add a number of dummy intervals to achieve this.

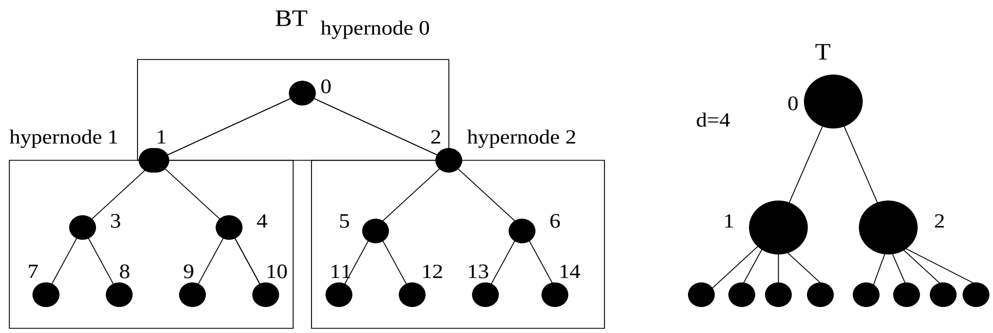

Figure 3: Trees BT and T for $d = 4$.

3.1 Parallel Base Tree of a Segment Tree

A direct approach to building in parallel the base tree BT would distribute its nodes evenly among the p processors, thus requiring fine-grained communication to access information of size $O(1)$ related to a particular node. Instead, we shall distribute evenly among the processors subtrees of BT of some predetermined depth $\lg d$, where d is dependent on n and p, so that coarse-grained communication patterns of size $O(d)$ can be utilized. The number of communication phases required to build or search a segment tree would also be reduced to $O(\lg n / \lg d)$. In order to achieve this we shall transform BT into a structure that is shallower, in terms of height, but richer, in terms of information stored at any node. We explain our approach in more detail below.

We coalesce a node v of BT at some level $\lg n - i \cdot \lg d$ and all its descendants up to level $\lg n - (i-1) \cdot \lg d$ into a "hypernode". We also coalesce $root(BT)$ and all its descendants up to level $\lg n - j \cdot \lg d$, where $j = \lfloor \lg n / \lg d \rfloor$, into another hypernode that will become the root of a tree of hypernodes. The children of a hypernode u are thus nodes of BT at distance $\lg d$

away from it in BT. The hypernode structure thus extracted from BT will be referred for clarity as T. The depth of T is then $\lceil \lg n / \lg d \rceil$. The number of leaves and internal nodes in BT is n and $n-1$ respectively. Each node v of BT stores $O(1)$ information (pointers to children and an $Int(v)$ interval). The number of leaves in T is the same, n, and its number of internal nodes is $N = (n/(d-1))(1 - 1/d^{\lfloor \lg n / \lg d \rfloor}) + 1 \leq n/(d-1) + 1 = O(n/d)$. The information associated with each hypernode u is a subtree of BT of height $\lg d$ that will be referred to as the *hypertree* $H(u)$ of u and its size is thus a factor of $O(d)$ larger than that of a node in BT as it contains the information associated with all the nodes of a subtree of BT of height at most $\lg d$. We shall call *vertices* the nodes of a hypertree to distinguish them from the hypernodes. This way, a node of BT becomes a hypernode of T or a vertex of a hypertree of T.

We thus maintain the base tree BT in the form of T by distributing among the processors hypernodes of T, which have a richer structure, rather than nodes of BT. A base tree BT and its corresponding tree T of hypernodes are depicted in Figure 3. Each node of BT is assigned a unique number in a top-to-bottom left-to-right fashion. Each hypernode of T as a node of BT also maintains this numbering. In addition, the hypernodes of T are also numbered. This way, a hypernode can be uniquely identified by its numbering in T or BT. In the algorithms to follow we shall choose d so that in the most interesting cases $d = (n/p)^{\alpha}$, where α is a constant bounded such that $0 < \alpha < 1$. We may assume, without loss of generality, that d is a power of two, since otherwise we could increase α (that effectively doubles d in the worst case) to get the desired d. There is a straightforward bottom-up way to build T on the BSP model that also preserves the structure of BT (including the *Int* intervals).

Each processor preallocates memory to store no more than $\lceil N/p \rceil$ hypernodes for a total space requirement of $O(n/p + d)$ (hypernode numbered i in T is allocated space at processor $i \bmod p$). Let $anc(v)$ denote in BT the ancestor of v, $\lg d$ levels up (ie its father in T). The algorithm to build T proceeds in $\lg n / \lg d$ phases.

Initially, the n elementary intervals are presented ordered (left-to-right due to sorting of the x-coordinates of the end-points) and evenly distributed among the processors. They form the leaves of T and BT and numbered as nodes of T and BT (e.g. the i-th leaf from the left is node $n-1+i$ in BT). The *Int* of a leaf is its elementary interval. Subsequently, by the completion of phase $j > 0$, all hypernodes of T that correspond to nodes of BT of level $\lg n - j \lg d$ are constructed.

In the following phase $j+1$, all newly formed hypernodes v_l in the previous phase with the same ancestor $u = anc(v_l)$ in BT send their newly formed intervals $Int(v_l)$ to the processor that has allocated memory for hypernode u. The communication time is $\max\{L, gO(n/(pd^{j-1}))\}$ as all the hypernodes of a particular level are evenly distributed among the processors. The processor storing hypernode u then builds a base tree (ie $H(u)$) of height $\lg d$ (or smaller, if u is $root(BT)$) with elementary intervals the d received intervals, stores this structure in the preallocated space and updates the pointers of the d children of u to point to the correct processor/node locations storing v_l. The computation time required for this step is $\max\{L, O(n/(pd^{j-1}) + d)\}$.

After the final phase, the information associated with the root of T is also broadcast to every other processor in time $T_{brd}^{O(d)}(p)$. The root is the only hypernode whose thus formed base tree may be of height less than $\lg d$ (ie. the space required to store it may be less than $\Theta(d)$).

In the remainder, we refer as $node\_info(u)$ the information associated with each hypernode u, that is $H(u)$, the Int intervals associated with it, and the Int interval of u. The following results can then be obtained.

Lemma 3.1 *A BSP algorithm that constructs n elementary intervals from r non-intersecting line segments in the plane requires time $O(n/p) + O(gn/p) + O(L) + T_{srt}(n,p)$, where $r \leq n \leq 4r+1$ and $T_{srt}(n,p)$ is the time required to sort n keys on p processors under the BSP model.*

Lemma 3.2 *A BSP algorithm that constructs a base tree T from n elementary intervals requires total time $T_c(n,d) = \lceil \lg n / \lg d \rceil O(L) + gO(n/p+d) + O(n/p+d) + T_{brd}^{O(d)}(p)$, where d is the degree of every hypernode of T except, possibly, of its root. The $node\_info$ of $root(T)$ is available to every processor, and the space allocated per processor is $O(n/p+d)$.*

3.2 Parallel Segment Tree Construction

After building the parallel base tree T, the construction of the segment tree is completed by the execution of a process that performs the parallel equivalent of Build_STree. The information that needs to be updated in line 2 of Build_STree depends on the particular problem we wish to solve. We first present a variant of Build_STree that will solve the *Counting Problem*. This routine will also be utilized in the solution of the other problems as well.

3.3 The Counting Problem

For the Counting problem, line 2 of Build_STree increments a counter $c(u)$ initialized to zero and associated with every vertex u of hypertree $H(v)$ and every hypernode v of T. The parallel version of Build_STree to solve the *Counting Problem* called BuildBSP_C_STree accepts as input r line segments that are evenly, though arbitrarily, distributed among the processors. Each segment will contain a field indicating the current hypernode/vertex visited by the segment. BuildBSP_C_STree consists of $\lceil \lg n / \lg d \rceil$ phases. From the discussion prior to the presentation of Build_STree, the set of query segments can grow up to $4r$ at any phase of Build_STree and subsequently, of BuildBSP_C_STree. We use the generic symbol m to refer to the size of the input segment set at some phase of the algorithm; in the first phase $m = r$, and thereafter, $m \leq 4r$. The number of leaves of the parallel base tree is n, where by construction $n \leq 4r + 1$.

The presented algorithm extends significantly the one for multi-way search presented in [9]; the differences arise from the fact that the tree is stored differently and the set of segments (in fact, pairs of a segment and the vertex it traverses) dynamically changes in each phase.

In the presentation below some definitions will be utilized. For a hypernode v of T we define the set of line segments visiting v to be the *thread* of v. If v is at level l of T, we may also refer to this thread as a thread at level l. A thread may be stored on a single processor; such a thread is called *exclusive*, otherwise it is called *shared*. The contiguous set of processors that will store a shared thread will form a *group* and the first processor (lower-numbered) of the group will be called its *group-leader*. Tree T will be stored in the form described earlier to achieve communication efficiency for the BSP algorithm to be presented. The performance of an arbitrary phase l, is summarized in the following proposition.

Proposition 3.1 *Let T be the parallel base tree of a segment tree on $r \leq n \leq 4r + 1$ elementary intervals constructed from r line segments. Let $d = o(n/p)$. Then, algorithm* BuildBSP_C_STree *constructs a segment tree (without segment lists) whose l-th phase, $0 \leq l < \lceil \lg n / \lg d \rceil$, requires computation and communication time given by*

$$C(m, n, r, p, l) = 4h \left(\max \{ \left\lceil \frac{m}{p} \right\rceil, \frac{r}{p} \} \right) + O\left(\frac{m}{p} + \frac{n}{p} + L \right) + O\left(C_{ppf}^{O(d)}(p) \right),$$

and

$$M(m, n, r, p, l) = O\left(g\frac{m}{p} + g\frac{n}{p} \right) + O(M_{ppf}^{O(d)}(p)),$$

where m is the number of segments to be inserted in the beginning of phase l ($m = r$ for $l = 0$), and $h = \lg d$, for $l > 0$, with $h = depth(root(T))$ for $l = 0$. The algorithm requires $O(n/p + m/p)$ space per processor.

Proof: Let $h = \lceil \lg n / \lg d \rceil$. Procedure BUILDBSP_C_STREE is described as a sequence of stages. The following *invariants* will be maintained at each phase $0 \le l < h - 1$.

(1) At the completion of phase $l - 1$ the line segments are sorted with respect to the T numbering of the leaf vertices (which are hypernodes of level l) of the hypertrees associated with the hypernodes of level $l - 1$ of T.

(2) At the completion of phase $l - 1$ all shared threads are fully identified and characterized by their group and group leader.

The structure the algorithm maintains is that for every line segment δ there exists a processor that contains it and the hypernode it is visiting. Let S denote the set of m input line segments. Each segment maintains a field that indicates the hypernode or hypertree vertex it is currently visiting. Each thread is associated with a hypernode v and *node_info*(v) contains all the auxiliary structure for v; sometimes we may write for simplicity *node_info* without quantifying v. As the number of internal hypernodes is $N \le n/(d-1) + 1 = O(n/d)$, and each processor stores $\lceil N/p \rceil$ of them, each processor uses $O(Nd/p + d) = O(n/p + d)$ space.

In phase 0 the line segments form a single shared thread of size $m = r$; by virtue of the construction of the base tree T, *node_info*$(root(T))$ is also available to every processor. The algorithm description below treats phase 0 as any other phase without taking this information into consideration; the result is that *node_info*$(root(T))$ will be obtained by the group leader (processor zero) and broadcast to the remaining processors, a redundant step. Note also that the depth of the hypertree of $root(T)$ may not be $\lg d$, as it is the case for every other hypertree of T. The generic term h takes this into consideration; $h = \lg d$ for all phases other than phase 0.

Procedure BUILDBSP_C_STREE (l, S)

begin

Stage 1 : For each thread residing in a processor, the *node_info*(v) structure of the hypernode v visited by the line segments of the thread is obtained.

[**Analysis of Stage 1:** Let s be the size of some thread $\bar{S}$ formed in the previous phase and whose segments visit hypernode v or the single thread associated with hypernode $root(T)$. Depending on s and the exclusivity of $\bar{S}$ and in order to minimize communication during the access of *node_info* information we employ various methods. Informally, if s is larger than d, the size of (order of magnitude) the associated *node_info* (cases 1-2 below), the latter information is obtained from the appropriate processor; otherwise (case 3 below), the thread is dispatched to the processor maintaining *node_info*. If $\bar{S}$ is held by processor $\bar{p}$ and computation load-balancing parameter ϵ is such that $\epsilon = O(\sqrt{d})$, one of the following branches is followed.

(1) read *node_info*	if $s \geq d/\epsilon$ and $\bar{S}$ is exclusive,
(2) read *node_info*	if $s \geq d/\epsilon$, $\bar{S}$ is shared, and $\bar{p}$ is group leader,
(3) send $\bar{S}$ to $\hat{p}$	if $s < d/\epsilon$,
(4) do nothing	otherwise,

where $\hat{p}$ is the unique processor storing *node_info* for the hypernode visited by thread $\bar{S}$. The first time this procedure is executed ($l = 0$) every processor holds at most $\lceil m/p \rceil$ segments. Thereafter, each processor may hold up to $\lceil 4m/p \rceil$ segments (but no more than $\lceil 4r/p \rceil$). Each processor also maintains at most $\lceil N/p \rceil + 1$ hypernodes each one holding $O(d)$ information (the additive 1 is due to $root(T)$). The number of *node_info* records each processor receives for Cases 1 and 2 is at most $\epsilon m/(dp) + 1$. Each processor, by way of Case 3 receives a set S^* of at most $s^* = \lceil N/p \rceil d/\epsilon$ additional line segments. Hence, this step takes $\max\{L, g\, O(\epsilon m/p + (n/p + d)/\epsilon)\}$. If $\bar{S}$ is shared, then $\bar{p}$ broadcasts *node_info*, obtained in the previous step, to all processors of its group. This step requires time at most $T_{ppf}^{O(d)}(p)$ by way of a segmented parallel prefix operation [18]. At the completion of this stage each processor holds at most $\lceil m/p \rceil + s^*$ segments.]

Stage 2 : Every processor, for each local segment δ visiting hypernode v invokes Build_STree(v, δ) on $H(v)$ with line 2 of the code updating $c(v)$.

[**Analysis of Stage 2:** All required information needed for Build_STree, that is, $H(v)$ contained in *node_info*, is locally available. At the end of this stage scalar values $c(w)$, for every vertex w of $H(v)$ will be updated and the number of pairs (segment, hypernode this segment visits in phase $l+1$) at the completion of this stage will be up to four times larger than that in the beginning of this stage, which was $\lceil m/p \rceil + s^*$.

Thus, the time required by this stage is $\max\{L, 4 \lg d (\lceil m/p \rceil + s^*)\}$ if $l > 0$ ($h = \lg d$) and $\max\{L, 4 \lg d' (\lceil m/p \rceil + s^*)\}$ otherwise ($h = \lg d'$), where $2d' - 1$ is the size of the hypertree of $root(T)$.

There is an imbalance of segments on processors effected by the term s^*. If $\epsilon = \omega(1)$ one-optimality can be maintained for the algorithm in hand but with a ϵ-wise increase in communication (communication time of stage 1). An alternative communication efficient way is to use the following load-balancing method. Since $s^* = \lceil N/p \rceil d/\epsilon$ and $N \leq n/(d-1)+1$, we get that $s^* \leq nd/((d-1)p\epsilon) + o(n/p)$. Since $n \leq 4r+1$, by choosing appropriate $4 < \epsilon < 8$, we can maintain $s^* \leq r/p$.

Let us denote set S^* on processor i by S_i^* and let its size be s_i^*. Let the remaining segments on processor i form set S_i. Segments in S_i^* need to be processed locally, those in S_i need not. We first load balance set $\cup_i S_i$ using parallel-prefix and regular routing operations of size at most $\max\{\lceil m/p \rceil, r/p\}$ and then, these segments fill the processors so that processor i receives $\max\{\lceil m/p \rceil, r/p\} - s_i^*$ of them; parallel-prefix and regular routing is required to achieve this. The load balancing scheme requires overall time $2\max\{L, O((m+r)/p)\} + 2\max\{L, O(g(m+r)/p)\} + O(T_{ppf}(p))$.

If we employ this method, ϵ is set $\epsilon = \Theta(1)$ for the remainder of this discussion. The execution time of the call to BUILD_STREE in this stage is then bounded by $\max\{L, 4h\max\{\lceil m/p \rceil, r/p\}\}$ then.]

Stage 3 : The updated in stage 2 *node_info* structures are routed back to their original storage area (reverse of stage 1 communication). For shared threads a parallel-sum operation implemented in terms of a parallel-prefix operation is initiated on the $c(v)$ values of the $H(u)$ hypertree of a hypernode u whose thread is shared, before the routing is effected by the leader processor of the group.

[**Analysis of Stage 3:** The time complexity of this step is, if the load-balancing step of stage 2 is ignored, $\max\{L, g\, O(\epsilon m/p + s^*)\} + T_{ppf}^{O(d)}(p)$. If the load-balancing scheme described in stage 2 has been performed its actions need to be reversed before stage 3 is performed; the time complexity of this reversal is an additional $\max\{L, O(g(m+r)/p)\}$ term. We note that the parallel sum operation updates the $c(v)$ values of $H(u)$ correctly as each processor assigned to the thread of u computed only a portion of the segment lists in its own copy of $H(u)$ communicated in Stage 1.]

Stage 4 : At the completion of stage 2 each segment includes information on the hypernode it will visit in the following phase. Processors sort segments within each thread with respect to the T number of this hypernode, which is the vertex of the last level of the hypertree accessed in stage 1; some segments of S may be absent after the completion of step 2. This stage maintains invariant 1 for phase $l+1$.

[**Analysis of Stage 4:** Let s be the size of some thread $\bar{S}$ as formed at the end of stage 2 before the load-balancing step, whose effects were reversed in Stage 3. The sorting operation is performed by one of several algorithms depending on s and the exclusivity of $\bar{S}$.

(1) RADIX_SORT if $s \geq \sqrt{d/\epsilon}$ and $\bar{S}$ is exclusive,

(2) PARALLEL RADIX_SORT if $s \geq \sqrt{d/\epsilon}$ and $\bar{S}$ is shared,

(3) SUB_QUADRATIC_SORT if $s < \sqrt{d/\epsilon}$.

If a radix-sort based algorithm is used the T number of a hypernode (leaf hypertree vertex) needs to be normalized in the range $[0, \ldots, d-1]$ before the sorting and restored afterwards; for shared threads this involves a broadcasting implemented in terms of a parallel prefix operation [18] initiated by the group leader that communicates the normalized numbers for the d hypernodes and a linear time scan per processor to update these numbers. The lower-indexed of the at most two processors that may share an $\bar{S}$ in Case 3 is to perform procedure SUB_QUADRATIC_SORT. The overall size of all threads of size greater than $\sqrt{d/\epsilon}$ on any one processor is at most $4(\lceil m/p \rceil + s^*)$; algorithms RADIX_SORT and PARALLEL_RADIX_SORT, since $\epsilon = O(\sqrt{d})$, require $O(1)$ iterations of sequential radix sort that sorts at least $\sqrt{d/\epsilon}$ integers in the range $[0, \ldots, d-1]$. The time for sorting all such threads is $O(\max\{L, g(m/p + s^*)\}) + O(\max\{L, m/p + s^*\}) + O(T_{ppf}^{O(d)}(p))$.

When SUB_QUADRATIC_SORT is applied, each thread of size less than $\sqrt{d/\epsilon}$ is sorted in $o(d/\epsilon)$ time. The total time required to sort all such threads $\bar{S}_i$ of size $s_i < \sqrt{d/\epsilon}$ on any particular processor is $\max\{L, o(\sum_i s_i^2)\} \leq \max\{L, o(\sum_i d/\epsilon)\}$. As $i \leq \lceil N/p \rceil$, $\sum_i s_i^2 \leq \lceil N/p \rceil d/\epsilon$. Thus, the total time required to sort all such threads is bounded above by $O(\max\{L, o((n/p + d)/\epsilon)\})$ and $O(\max\{L, o(g(n/p + d)/\epsilon)\})$ for computation and communication respectively.]

Stage 5 : Each processor returns to the originating processors the threads it received in stage 1. A load-balancing operation is performed.

[**Analysis of Stage 5:** Since the total size of all the threads that any single processor sent or received in stage 1 is at most $\lceil m/p \rceil$ and s^* respectively, the time complexity of this stage is bounded above by $O(\max\{L, 4g(m/p + s^*)\})$ as the size of each original thread may have been increased by a factor of 4. The load-balancing operation requires time $\max\{L, O(m/p)\} + \max\{L, O(gm/p)\} + T_{ppf}(p)$.]

Stage 6 : Each processor prepares for the following phase by characterizing, by group and group leader, the threads induced in the current phase. This stage updates invariant 2 for phase $l+1$.

[**Analysis of Stage 6:** Since the segments are sorted with respect to hypernodes of the following level of T each processor can determine all of its exclusive threads by simply scanning the segments in its local memory. Moreover, if there are shared threads, all of them can be identified and characterized by communicating a constant number of messages and by performing a segmented parallel-prefix operation. After completion of the parallel prefix operation each processor knows for each of its shared threads its group and group leader. When this stage is executed every processor stores at most $\min\{\lceil 4m/p \rceil, \lceil 4r/p \rceil\}$ segments and at most two shared threads. Thus, the cost of this stage is bounded above by $\max\{L, O((m+r)/p)\} + O(T_{ppf}(p)) + \max\{L, O(g)\}$.]

end

We therefore obtain that the time complexity, over all stages, of procedure MULTI_WAY SEARCH is (h is $\lg d$ for all phases other than phase 0, and for phase 0 it is the actual depth of $H(root(T))$)

$$4h\left(\max\{\left\lceil\frac{m}{p}\right\rceil, \frac{r}{p}\}\right) + O\left(\frac{m}{p} + \frac{n}{p} + \frac{r}{p} + L\right) + O\left(C_{ppf}^{O(d)}(p)\right),$$

and

$$O\left(g\frac{m}{p} + g\frac{n}{p} + g\frac{r}{p} + L\right) + O(M_{ppf}^{O(d)}(p)),$$

as in the beginning of phase 0, it was $m = r$, at each phase $m \leq 4r$, and load-balancing was used with $\epsilon = \Theta(1)$ ($\epsilon < 8$). ∎

If we sum up the contributions of all the phases and note that in the beginning of phase 0, it was $m = r$ and thereafter $m \leq 4r$, the following proposition is derived.

Proposition 3.2 *Let T be the parallel base tree of a segment tree on $r \leq n \leq 4r+1$ elementary intervals constructed from r line segments. Let $d = o(n/p)$. Then, algorithm* BUILDBSP_C_STREE *constructs a segment tree (without segment lists) in computation and communication time given by*

$$C_{C\_Build}(n, r, p) = 4\lg n\left(\frac{r}{p}\right) + \frac{\lg n}{\lg d}O\left(\frac{n}{p} + \frac{r}{p} + L\right) + \frac{\lg n}{\lg d}O\left(C_{ppf}^{O(d)}(p)\right),$$

and

$$M_{C\_Build}(n, r, p) = \frac{\lg n}{\lg d}O\left(g\frac{n}{p} + g\frac{r}{p} + L\right) + \frac{\lg n}{\lg d}O(M_{ppf}^{O(d)}(p)).$$

The algorithm requires $O(n/p + r/p)$ space per processor. The running time of the algorithm (communication and computation) is denoted by $T_{C\_Build}(n, r, p)$.

By substituting appropriate values for the parameters in proposition 3.2 a variety of bounds on the run-time of algorithm BUILDBSP_C_STREE can be obtained. The following is one such bound.

Theorem 3.1 *Let T be a segment tree on $r \leq n \leq 4r+1$ elementary intervals constructed from r line segments. Let $d = o(n/p)$. Then, algorithm* BuildBSP_C_STree *for constructing a segment tree (without allocating the segment lists) such that for all n, p, L and t, where $t = \min\{O(L/g), p\}$ and $L = o(n \lg d/(p \log_t p))$ has, $\pi = 1 + o(1) + O(1/\lg d)$, and if $g = o(\lg d)$ then also, $\mu = o(1)$. The algorithm requires $O(n/p)$ space per processor.*

Proof: For computation time, by Proposition 3.2, after substituting for $C_{ppf}^{O(d)}(p)$, we get that

$$C_{C\_Build}(n,r,p) \quad \leq \quad 4\left(\frac{r \lg n}{p}\right) + O(\frac{\lg n}{\lg d}(\frac{n}{p} + O(d)) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L) + O(\frac{\lg n}{\lg d} \lg_t (p/d)t).$$

For $d = O(n/p)$ and since by choice of t, $t = O(L/g)$, we further get that

$$C_{C\_Build}(n,r,p) \quad \leq \quad 4\left(\frac{r \lg n}{p}\right) + O(\frac{\lg n}{\lg d}\frac{n}{p}) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L).$$

As the sequential running time is proportional to $O(4r \lg n)$, the expression for π in theorem 3.1 follows. Similarly,

$$M_{C\_Build}(n,r,p) \quad \leq \quad O(\frac{\lg n}{\lg d}(\frac{n}{p} + O(d))g) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L) + O(\frac{\lg n}{\lg d} \lg_t (p/d)O(tg)).$$

For $d = O(n/p)$ and since by choice of t, $t = O(L/g)$, we further get that

$$M_{C\_Build}(n,r,p) \quad \leq \quad O(\frac{\lg n}{\lg d}\frac{n}{p}g) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L).$$

The expression for μ in theorem 3.1 then follows after dividing with the sequential running time.

■

3.3.1 Query satisfaction

We have built in the previous section a segment tree that will be used for the satisfaction of the queries of the *Counting Problem*, where given a set S of $m \geq n$ queries, for each query q the number of segments intersected by a ray starting at q is computed (direction is the y axis). A segment-counter field will be attached to each query and will eventually store this counting information.

The query problem resembles the one in [9] and for $m \geq n$, it can be solved by an one-optimal algorithm. Algorithm BuildBSP_C_STree can be modified so that calls to Build_STree are replaced by calls to QUERY_STREE and each time a query visits node w it increments its segment-counter field by $c(w)$. It is also noted that S never grows in size (as is the case in BUILDBSP_C_STREE) thus a factor of four can be removed from the runtime. Details of the algorithm follow. For the Counting problem, line 1 of QUERY_STREE adds $c(u)$ to the value of the hit-counter of query q and is initialized to zero. The parallel version of QUERY_STREE to solve the *Counting Problem* is called QUERYBSP_C_STREE and accepts as input m queries that are evenly, though arbitrarily, distributed among the processors. Each query will also contain a field indicating the current hypernode/vertex visited by the query. We are going to use the same terminology employed in BUILDBSP_C_STREE. The performance of an arbitrary phase l, is summarized in the following proposition.

Proposition 3.3 *Let $S(T)$ be the parallel segment tree on n elementary intervals constructed by* BUILDBSP_C_STREE. *Let $d = o(n/p)$ be the size of a hypertree in $S(T)$.* QUERYBSP_C_STREE *then, satisfies $m \geq n$ evenly distributed queries whose l-th phase, $0 \leq l < \lceil \lg n / \lg d \rceil$, requires computation and communication time given by*

$$C(m, n, p, l) = h\left(\frac{m}{p}\right) + O\left(\frac{m}{p} + \frac{n}{p} + L\right) + O\left(C_{ppf}^{O(d)}(p)\right),$$

and

$$M(m, n, p, l) = O\left(g\frac{m}{p} + g\frac{n}{p} + L\right) + O(M_{ppf}^{O(d)}(p)),$$

where $h = \lg d$, for $l > 0$ and $h = depth(root(T))$ for $l = 0$. The algorithm requires $O(n/p + m/p)$ space per processor.

Proof: Procedure QUERYBSP_C_STREE is described as a sequence of stages. The following *invariants* will be maintained at each phase $0 \leq l < h - 1$.

(1) At the completion of phase $l-1$ the queries are sorted with respect to the T numbering of the leaf vertices of the hypertrees associated with the hypernodes of level $l-1$ of T they will be visiting in the following phase.

(2) At the completion of phase $l-1$ all shared threads are fully identified and characterized by their group and group leader.

The structure the algorithm maintains is that for every query q there exists a processor that contains it and the hypernode it is visiting. Let S denote the set of m input queries. Each query, besides the hit-counter maintains a field that indicates the hypernode or hypertree vertex it is currently visiting.

We briefly describe the stages of QUERYBSP_C_STREE without providing detailed explanation of the various stages, since it can inferred from the discussion of algorithm BUILDBSP_C_STREE.

Procedure QUERYBSP_C_STREE (l, S)

begin

Stage 1 : For each thread residing in a processor, the *node_info*(v) structure of the hypernode v visited by the queries of the thread is obtained.

Stage 2 : Every processor, for each local query q visiting hypernode v invokes QUERY_STREE(v, δ) on $H(v)$ with line 2 of the code adding $c(v)$ to the contents of the hit-counter of q.

Stage 3 : The updated in stage 2 *node_info* structures are routed back to their original storage area (reverse of stage 1 communication).

Stage 4 : At the completion of stage 2 each query includes information on the hypernode it will visit in the following phase. Processors sort queries within each thread with respect to the T number of this hypernode, which is the vertex of the last level of the hypertree accessed in stage 1. This stage maintains invariant 1 for phase $l+1$.

Stage 5 : Each processor returns to the originating processors the threads it received in stage 1.

Stage 6 : Each processor prepares for the following phase by characterizing, by group and group leader, the threads induced in the current phase. This stage updates invariant 2 for phase $l+1$.

end■

The following proposition describes the performance of QUERYBSP_C_STREE over all phases.

Proposition 3.4 *Let $S(T)$ be the parallel segment tree on n elementary intervals constructed by* BUILDBSP_C_STREE. *Let $d = o(n/p)$ be the size of a hypertree in $S(T)$.* QUERYBSP_C_STREE *then, satisfies $m \geq n$ evenly distributed queries in computation and communication time given by*

$$C_{C\_Query}(m, n, p) = \lg n \left(\frac{m}{p}\right) + \frac{\lg n}{\lg d} O\left(\frac{n}{p} + \frac{m}{p} + L\right) + \frac{\lg n}{\lg d} O\left(C_{ppf}^{O(d)}(p)\right),$$

and

$$M_{C\_Query}(m, n, p) = \frac{\lg n}{\lg d} O\left(g\frac{n}{p} + g\frac{m}{p} + L\right) + \frac{\lg n}{\lg d} O(M_{ppf}^{O(d)}(p)).$$

The algorithm requires $O(n/p + m/p)$ space per processor. Let the running time of the algorithm be denoted by $TM_{C\_Query}(m, n, p)$.

By substituting appropriate values for the parameters in proposition 3.3 a variety of bounds on the run-time of algorithm QueryBSP_C_STree can be obtained. The following is one such bound.

Theorem 3.2 *Let $S(T)$ be the parallel segment tree on n elementary intervals constructed by algorithm* BuildBSP_C_STree. *Let $d = o(n/p)$ be the size of a hypertree in $S(T)$. Then, algorithm* QueryBSP_C_STree *for any set of $m \geq n$ evenly distributed queries and for all n, p, L and t, where $t = \min\{O(L/g), p\}$ and $L = o(n \lg d/(p \log_t p))$ has, $\pi = 1 + o(1) + O(1/\lg d)$, and if $g = o(\lg d)$ then also, $\mu = o(1)$. The algorithm requires $O(n/p)$ space per processor.*

Proof: For computation time, by Proposition 3.4, after substituting for $C_{ppf}^{O(d)}(p)$, we get that

$$C_{C\_Query}(m,n,p) \leq \left(\frac{m \lg n}{p}\right) + O(\frac{\lg n}{\lg d}(\frac{m}{p} + O(d)) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L) + O(\frac{\lg n}{\lg d} \lg_t (p/d)t).$$

For $d = O(n/p)$ and since by choice of t, $t = O(L/g)$, we further get that

$$C_{C\_Build}(m,n,p) \leq \left(\frac{m \lg n}{p}\right) + O(\frac{\lg n}{\lg d}\frac{m}{p}) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L).$$

As the sequential running time is proportional to $O(m \lg n)$, the expression for π in theorem 3.2 follows. Similarly,

$$M_{C\_Build}(n,r,p) \leq O(\frac{\lg n}{\lg d}(\frac{m}{p} + O(d))g) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L) + O(\frac{\lg n}{\lg d} \lg_t (p/d)O(tg)).$$

For $d = O(n/p)$ and since by choice of t, $t = O(L/g)$, we further get that

$$M_{C\_Build}(n,r,p) \leq O(\frac{\lg n}{\lg d}\frac{m}{p}g) + O(\frac{\lg n}{\lg d} \lg_t (p/d)L).$$

The expression for μ in theorem 3.2 then follows after dividing with the sequential running time.

■

3.4 The First Hit Problem

A straightforward sequential solution for the first-hit problem first builds a segment list with sorted segment lists in $O(n \lg n)$ time (a factor of $\lg n$ faster than the sequential algorithm implied from the parallel segment-tree construction method that is outlined below). Subsequently, each query performs a binary search on the base tree of the segment tree and on each node visited during that search, a second binary search on the sorted segment list I of that node is also performed. The search time per query is thus $O(\log^2 n)$.

We next sketch a solution for the First-hit problem that achieves optimal performance if compared to the straightforward non-optimal algorithm. In the parallel case, $m \geq n$ queries will be evenly distributed among the processors as is T (and thus BT). Sequential search time of $O(m \log^2 n)$ absorbs the costs of the inefficient $O(n \log^2 n)$ construction method that builds a segment tree with sorted segment lists.

Procedure BUILDBSP_FH_STREE.

Step 1. The first step in the solution involves the construction of T by invoking a call to BUILDBSP_C_STREE. As a result for every hypernode and vertex of T the segment list size $c(w)$ is obtained. Therefore the size of the segments list at each hypernode (global information used for allocation of space for the lists of a hypertree) and vertex (local information used for allocation of space for the lists within a hypernode) of T becomes available.

The cost of this step is the cost of BUILDBSP_C_STREE.

Step 2. A parallel prefix operation on the segment list size values $c(w)$, for every hypernode or vertex w, is initiated. The order of the prefix operation is dictated by the BT numbering of the hypernodes/vertices. As a result, for each such vertex w the total size of all $I(u)$ lists of all vertices/hypernodes $u < w$ is computed. At the same time, the total size of all segment lists, also becomes available. Let this size be k.

The cost of this stage consists of the cost of an initial routing operation to order vertices according to their BT numbering, i.e. $\max\{L, g(2n-1)/p\}$, a parallel prefix operation with associative operator addition that requires time $\max\{L, (2n-1)/p\}$ for a local (sequential prefix) computation per processor followed by a parallel prefix operation on the p partial sums (one per processor) that requires time $T_{ppf}(p)$ and a final step, where partial results in the first phase are updated, that requires additional $\max\{L, (2n-1)/p\}$ time. The size of k is then broadcast to all processors for additional time of $M_{brd}(p) \le T_{ppf}(p)$ (if implemented through parallel prefix).

Step 3. Each processor allocates space $\lceil k/p \rceil$ for the storage of the segment lists and each hypernode u maintains the results of the parallel prefix operation related to all its $H(u)$ vertices. As $k = O(n \lg n)$, $O(n \lg n/p)$ space per processor is allocated.

The time required for this step is $\max\{L, O(n \lg n/p)\}$ for space allocation and initialization and $\max\{L, O(gn/p)\}$ for information maintenance (storing the results of parallel prefix in *node_info*).

Step 4. A second call to BuildBSP_C_STree is issued. During this call, in stage 2, line 2 of Build_STree is modified so that the $I(w)$ structures are created/updated instead of the segment list sizes $c(w)$.

We first note that *node_info* structures will not transfer the $I(w)$ lists. Whenever a segment list $I(w)$ or a portion of such a list (for shared threads that reside in more than one processors, $I(w)$ created in say processor l reflects the state of $I(w)$ only for the segments of the thread visiting w that reside in l) is created in some processor, it remains stored in that processor until the last phase of BuildBSP_C_STree.

Although it is not necessary, the parallel sum operation of stage 3 (which becomes redundant, as segment list sizes are already available) can be replaced by a parallel prefix operation (over associative operator addition) on the $c(w)$ values that reflect the size of $I(w)$ for the segments residing on a specific processor. If these parallel prefix results are added to the ones of step 2, the exact location of each sublist of $I(w)$ (due to thread fragmentation) in the space allocated in step 3 can also be computed.

The cost of this step is the cost of BuildBSP_C_STree. We note again that segment list/sublist transmission occurs after all phases are completed for a total of $O(gn \lg n/p)$ communication time.

Step 5. At the completion of step 4, the memory of each processor stores pairs of the form (w, s), where w is a vertex and s is a segment belonging to $I(w)$. Note that segment/vertex pairs are sorted with primary key w (numbering in BT). A global sorting operation is performed anyway that sorts pairs with primary key w and secondary key s. Segments are sorted with respect to the "up"/"down" order of lines segments. Let x_w be the x-coordinate of the left endpoint of $Int(w)$. Two segments in $I(w)$ are compared by comparing the corresponding y coordinates of the intersection of line $x = x_w$ with each segment.

The cost of this step is the cost of sorting at most $4n \lg n$ segments ie $T(4n \lg n, p)$.

Step 6. The segment lists/sublists are transmitted to their destination (are allocated in step

3).

This step takes $\max\{L, O(gn\lg n/p)\}$ communication time.

The following is thus shown.

Proposition 3.5 *Let T be the base tree of a segment tree on $r \le n \le 4r+1$ elementary intervals constructed from r line segments. Let $d = o(n/p)$ be the size of a hypertree in T. Then,* BuildBSP_FH_STree *builds a segment tree with sorted segment lists in time,*

$$\begin{aligned} T_{FH\_Build}(n,r,p) &= 2T_{C\_Build}(n,r,p) + O(L) + T(4n\lg n, p) + O(gn\lg n/p) + \\ &\quad 2T_{ppf}(p) + O(n/p). \end{aligned}$$

The algorithm requires $O(n\lg n)$ space per processor.

A theorem similar to Theorem 3.2 can be derived. Its optimality depends on the sorting algorithm that is used. If one of [10, 12, 13] is used then one optimality can be claimed for BuildBSP_FH_STree. These sorting algorithms however are not fully scalar; they require that $n/p = \lg^{1+\alpha} n$, where $\alpha > 0$ or $\alpha > 1$ depending on the algorithm. If however, the fully scalar algorithm of [15] is used, BuildBSP_FH_STree becomes c-optimal for some large constant c.

3.4.1 Query satisfaction

We utilize the query algorithm QueryBSP_C_STree to solve the first-hit problem. We assume that $m \ge n$ queries are evenly, though arbitrarily distributed, among p processors.

Procedure QueryBSP_FH_STree.

Step 1. Associated with each query is a list of size $\lg n$. This list records the path a query follows during the search i.e. the execution of algorithm QueryBSP_C_STree. Each element of such a list is of the form (q, w), where q is a query and w is a hypertree vertex traversed by it. When a list element is generated by a query, it is stored locally in the processor that generated it, and only transmitted at the completion of the latest phase of QueryBSP_C_STree to the processor originally storing q, before phase 0.

The generation of such lists does not affect the running time of step 1 which is expressed in terms of the running time of QueryBSP_C_STree, ie $T_{C\_Query}(m, n, p)$.

Step 2. A balanced d-ary tree on $O(n\lg n)$ leaves which are the pairs (w, s) that were generated by BuildBSP_FH_STree is built the same way the T tree was generated from the elementary intervals before BUILDBSP_C_STREE. We note that the constructed tree is similar to T but without hypertrees or $c(w)$ lists and in place of $Int(u)$ a key of the form (w, s) is maintained.

The running time of this step is $T_c(n, d)$.

Step 3. Subsequently, the m queries are searched on this tree by invoking QUERY_C_STREE. Due to the simplified structure of the tree as explained in step 2 the search process is much simpler. The input queries are the query pairs (q, w). When a query pair (q, w) is compared to a node (w, s) of that tree primary key is w and secondary key is the result of the comparison between q and s according to a "below"/"above" relation between the two. The result of a query (q, w) is thus a leaf (w, s_q), the lowest segment in $I(w)$ hit by a ray starting at q in direction parallel to the y-axis.

The running time of this step is $T_{C\_Query}(m \lg n, n \lg n, p)$.

Step 4. The partial results obtained in Step 3 are transmitted to the processor location occupied by the query pairs in the beginning of QUERYBSP_FH_STREE. This way, the $\log n$ results of query q are adjacent. A linear segmented scan (per query) per processor followed by a segmented parallel prefix (segmented parallel minimum) identifies within each query the segment first hit by a query (the one whose intersection with a ray starting from the query and parallel to the y axis has the lowest y coordinate).

The running time of this step is $\max\{L, gO(m \lg n/p)\} + \max\{L, O(m \lg n/p)\} + T_{ppf}(p)$.

The following proposition summarizes the obtained result.

Proposition 3.6 *Let $S(T)$ be the parallel segment tree on n elementary intervals constructed by algorithm* BUILDBSP_FH_STREE. *Let $d = o(n/p)$ be the size of a hypertree in $S(T)$. Then,* QUERYBSP_FH_STREE *for satisfying $m \geq n$ evenly distributed queries has running time given by*

$$\begin{aligned} T_{FH\_Query}(m, n, p) &= T_{C\_Query}(m, n, p) + T_{C\_Query}(m \lg n, n \lg n, p) + \\ &\quad \max\{L, gO(m \lg n/p)\} + \max\{L, O(m \lg n/p)\} + T_{ppf}(p) + T_c(n, d). \end{aligned}$$

A theorem similar to theorem 3.2 can be claimed.

Theorem 3.3 *Let $S(T)$ be the parallel segment tree with sorted segment lists on n elementary intervals constructed by algorithm* BUILDBSP_FH_STREE. *Let $d = o(n/p)$ be the size of a hypertree in $S(T)$. Then, algorithm* QUERYBSP_FH_STREE *for any set of $m \geq n$ evenly distributed queries and for all n, p, L and t, where $t = \min\{O(L/g), p\}$ and $L = o(n \lg d/(p \log_t p))$ has, $\pi = 1 +$*

$o(1) + O(1/\lg d)$, and if $g = o(\lg d)$ then also, $\mu = o(1)$. The algorithm requires $O(n \lg n/p)$ space per processor.

We finally note that the First-Hit algorithm leads to a solution of the bichromatic line segment intersection problem [7] with the same performance in terms of π and μ as that of algorithm QueryBSP_FH_STree.

4 Acknowledgements

This work was supported in part by NJIT SBR Grant 421350. Some preliminary work was performed while the author was with the University of Oxford, Oxford UK, and supported in part by EPSRC under grant GR/K16999.

References

[1] M. J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J. Tsay. Multisearch techniques for implementing data structures on a mesh-connected computer. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures,* pp. 204-214, 1991.

[2] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the Annual European Symposium on Algorithms*, Lecture Notes in Computer Science, Springer-Verlag, 1995.

[3] M. de Bergh, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry: Algorithms and Applications. Springer-Verlag Berlin Heidelberg, 1997.

[4] A. Chan, F. Dehne and A. Rau-Chaplin. Coarse grained parallel next element search. In *Proceedings of the 11-th International Parallel Processing Symposium*, Geneva, 1997.

[5] A. Chan, F. Dehne and A. Rau-Chaplin. Coarse grained parallel geometric search. Journal of Parallel and Distributed Computing, 57(2), 1999, 224-235.

[6] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing,* 8:367-375, 1990.

[7] O. Devillers and A. Fabri. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. In Proceedings of the Workshop on Algorithms and Data Structures (WADS '93), Lecture Notes in Computer Science, pp. 277-288, Springer-Verlag, 1993.

[8] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267, Academic Press, 1994.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, May 1996. An extended abstract appeared in *Proceedings of EURO-PAR'96*, Lyon, France, LNCS 1124, Springer-Verlag, August 1996.

[10] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Primitive operations on the BSP model. Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996.

[12] A. V. Gerbessiotis and C. J. Siniolakis. Efficient Deterministic Sorting on the BSP Model. *Parallel Processing Letters*, Vol. 9 No 1 (1999), pp 69-79, World Scientific Publishing Company.

[13] A. V. Gerbessiotis and C. J. Siniolakis. A randomized sorting algorithm on the BSP model. In *Proceedings of the 11-th International Parallel Processing Symposium*, Geneva, Switzerland, IEEE Press, April 1997.

[14] A. V. Gerbessiotis and C. J. Siniolakis. Ordered h-Level Graphs on the BSP Model. Journal of Parallel and Distributed Computing, 1998. *Journal of Parallel and Distributed Computing,* 49:98-110, Academic Press, 1998.

[15] M. T. Goodrich. Communication-Efficient Parallel Sorting. *Proceedings of the 26-th ACM Symposium on the Theory of Computing*, ACM Press, 1996.

[16] B. H. H. Juurlink and H. A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters,* 58:303-310, 1996.

[17] D. E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, Reading, 1973.

[18] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, California, 1991.

[19] F. P. Preparata and M. I. Shamos. Computational Geometry: An Introduction. Springer Verlag, New York, 1985.

[20] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM,* 33(8):103-111, August 1990.

[21] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science,* (J. van Leeuwen, ed.), North Holland, 1990.