Programming Research Group

DIRECT BULK-SYNCHRONOUS PARALLEL ALGORITHMS IN COMPUTATIONAL GEOMETRY

Constantinos J. Siniolakis

PRG-TR-10-96

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

The objective of this report is a unified investigation of a wide range of computational geometry algorithms and the design of Bulk-Synchronous Parallel algorithms that behave better than the immediate simulations of the corresponding PRAM algorithms. We present a methodology for constructing parallel computational geometry algorithms that are transportable, i.e. architecture independent algorithms capable of delivering scalable performance for a multiplicity of parallel architectures. The method we describe is based on the Bulk-Synchronous Parallel (BSP) model of computation, which abstracts the characteristics of a parallel machine into three numerical parameters p, g, and L, that quantify, respectively, processors, communication throughput to computation throughput, and synchronization period. These parameters, together with the problem size, are used to measure the performance, and, consequently, the transportability of geometric algorithms across parallel machines having different values of these parameters. In this report we theoretically analyze the efficiency with which several important geometric computations can be performed on bulk-synchronous parallel architectures. The computational problems considered include sorting, merging, bucketing, searching, intersections, convex hulls, maxima, visibility, proximity, voronoi diagrams, and geometric optimization. An important distinction between the approach that we consider in this report and the previous approaches is that our approach does not depend upon any particular choice of parallel system family but is universally applicable to any parallel machine (i.e., distributed memory system, shared memory multiprocessor, network of workstations).

[0]Part of this report was submitted as a qualifying dissertation to support an application for transfer of status from a Probationary Research Student to a Student for the Degree of Doctor of Philosophy at the University of Oxford

1 Introduction

Parallel Computational Geometry [3, 13, 56, 57, 96, 143, 163, 177] is a branch of computer science concerned with the study of designing and analyzing efficient (practical being of great importance nowadays) algorithms to treat computational problems dealing with collections of n Euclidean objects, i.e. points and lines in Euclidean space, on a parallel computer with p processors. This young discipline has attracted enormous research interest - mostly in the past decade - and has grown to a mature body of knowledge. This wealthy class of problems arises naturally, not only in the obvious application areas such us computer graphics, computer aided design (CAD), image analysis, pattern recognition and classification, robotics, but also in statistics, management science, operations research, and database search.

There is an affluence of sequential and parallel algorithms - mostly[1] Parallel Random Access Machine (PRAM) algorithms - for computational geometry problems, developed mainly over the last fifteen years. The PRAM [66, 69, 101, 102, 168, 174] is a synchronous model of parallel computation in which processors communicate via shared memory. It consists of some number of shared memory cells and processors. Each processor is a Random Access Machine (RAM) with a private local memory. During every step of computation, a processor may read from one shared memory cell into its registers, perform a local operation, or write to one shared memory cell. Different processors may execute different operations during a step and may make reference to their own index. Reads, local operations, and writes are viewed as occurring during three separate phases. The PRAM model provides an abstraction that strips away considerations such as communication latency, block transfers, memory and network conflicts during routing, bandwidth of interconnection networks [151], reliability, interprocessor communication, memory management, and synchronization. For this reason, algorithm design is simplified and PRAM algorithms [18, 38, 43, 44, 45, 77, 102, 143, 163] can be readily adapted to run on a variety of parallel machines. As a consequence, a considerable body of computational geometry fine grained PRAM algorithms has been discovered in the past several years, and a number of powerful techniques for designing such algorithms have been identified [101, 143, 163].

Consequently, theoretical work for computational geometry has so far focused on the case $\frac{n}{p} = O(1)$ (*fine grained parallelism*). However, for parallel geometric algorithms to be of any practical relevance, such algorithms must be *scalable*, that is, they must be applicable and efficient for an ample range of ratios $\frac{n}{p}$. As noted in [56], most existing (exciting) parallel computers (i.e. the Intel Paragon, Intel iPSC/860, CM-5, and IBM SP-2) consist of a set of p state of the art processors, each with considerable local memory and computing power, connected to some interconnection network (i.e. mesh, hypercube, fat tree). For such machines, the ratio $\frac{n}{p}$ is substantially larger than $O(1)$. Therefore, the design of scalable parallel algorithms is registered as one major goal in the 1992 *Grand Challenges* report [84].

Yet, only little theoretical work has been done for designing scalable parallel algorithms for computational geometry [57]. Notice that, in an idealized shared

[1] Someone may be tempted to argue that *all* parallel computational geometry algorithms are PRAM algorithms.

memory model such as a PRAM, trading time for processors is always possible - if there exists an optimal fine grained parallel algorithm, then, at least from a purely theoretical perspective, the problem is trivial:[2] Brent's scheduling principle secures an optimal simulation for any ratio $\frac{n}{p}$, i.e., if a problem can be solved in time T with p processing units, then, the same problem can be solved in $p^{\epsilon}T$ time with $p^{1-\epsilon}$ processors for any $0 \geq \epsilon \geq 1$.

Nevertheless, since the idealized PRAM model fails to reflect constraints in the real world (i.e. communication latency, bandwidth of interconnection networks, interprocessor communication and synchronization), its practical importance is limited. In reality most problems do not as yet have fine grained algorithms with linear speedup, or such linear speedup algorithms are impossible due to bandwidth or diameter limitations [57]. In fact, it is common knowledge that many optimal fine grained parallel computational geometry algorithms perform miserably in practice. This phenomenon may be attributed to the fact that most of the complexity of parallel computing is due to the difficulty in communication and synchronization rather than in computation (and obviously this problem will proliferate as the number of available processors increases). As noted in [5], researchers have recognized this issue and have investigated the communication complexity of special purpose parallel machines and VLSI chips [1, 7, 95, 130, 161, 162, 175]. Communication complexity is an equally important issue for general purpose parallel machines [115]. However, except for a few papers (i.e. [5, 131, 132]), which analyze communication complexity issues in rather abstract terms, a limited amount of research was directed towards the architecture-independent analysis of the communication complexity of parallel algorithms.

The purpose of this report is a unified investigation of a wide range of algorithms in this area, and the design of a few techniques that are useful for a better understanding of the communication requirements of parallel machines. The computational model adopted is the Bulk-Synchronous Parallel (BSP) model of computation, which deals explicitly with the notion of communication and synchronization among computational threads. The BSP model was originally proposed by Leslie Valiant [165] and subsequently addressed in [27, 115, 116] as a bridge between software and hardware, i.e. architecture independent software capable of delivering scalable performance for a wide class of applications on a variety of parallel computers [165].

In this report, we present, for a variety of problems, BSP algorithms with speedups considerably faster (approximately linear) than what can be obtained through the standard virtual processor simulation of fine grained algorithms (that

[2] In [83], it is shown that it is not always possible to trade time for processors in a fixed degree network because any permutation of n data elements on a p-processor, bounded degree interconnection network requires $\Omega(\frac{n}{p}\log p)$ time. In the same vein, any permutation algorithm for n data items on a BSP p-processor machine requires $g\frac{n}{p}$ time (where 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). Thus, in contrast to the idealized PRAM model, the BSP model is a more restrictive model of parallel computation and in order to understand complexity issues for the BSP model, it seems necessary to understand the following constraints that are inherent in this model: communication and synchronization (for a detailed discussion of this issue see [27, 74, 115, 116, 117, 165, 167]).

is, for $\frac{n}{p}$ larger than $O(1)$). Indeed, experiments with some of the algorithms presented in this report show that scalable parallel algorithms designed for $\frac{n}{p}$ larger than $O(1)$ seem to have considerable practical relevance. The idea behind most of the algorithms can be summarized as follows [56]. Partition the global problem of n data elements into p subproblems of equal size $O(\frac{n}{p})$. Each processor solves a constant number of such subproblems, and, eventually, by combining the $O(1)$ solutions of its $O(\frac{n}{p})$ size subproblems, each processor determines its $O(\frac{n}{p})$ size share of the global solution. The above is necessarily an oversimplification. The actual algorithms will do more than just those local (sequential) solutions. The major challenge lies in formulating a partitioning strategy that will ensure that the algorithm will terminate after $O(1)$ global rounds.

The important distinction between the approach that we consider in this report and the previous approaches is that our approach does not depend upon any particular choice of parallel system family but is universally applicable to any parallel machine (i.e., distributed memory system, shared memory multiprocessor, network of workstations).

2 Geometric Preliminaries and Model of Computation

2.1 Preliminaries and Notation

For the past twenty years the analysis and design of computer algorithms has been one of the most absorbing subjects in computer science. The fundamental work of Knuth [104] has originated an effort to bring order and uniformity to a rich compilation of otherwise isolated results. Subsequent works [6, 172] have further strengthen the theoretical foundations. The scope of this section is just to introduce the adopted concepts and notation.

2.1.1 Asymptotic Analysis

Knuth has manifested a notational methodology - usually referred to as *asymptotic analysis* - that differentiates properly between upper and lower bounds, which we shall adopt [106].

- $O(f(n))$ denotes the set of all functions $g(n)$ such that there exist two (unspecified) positive constants c and N with $g(n) \leq cf(n)$ for all $n \geq N$.
- $\Omega(f(n))$ denotes the set of all functions $g(n)$ such that there exist two (unspecified) positive constants c and N with $g(n) \geq cf(n)$ for all $n \geq N$.
- $\Theta(f(n))$ denotes the set of all functions $g(n)$ with $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$.

- $o(f(n))$ denotes the set of all functions $g(n)$ such that for *all* positive constants c there is a positive constant N with $g(n) \leq cf(n)$ for all $n \geq N$.

Obviously, the "order of" notation conceals multiplicative constants, therefore, a complexity result secures its full validity only for sufficiently large problem sizes. Hence, it is not uncommon that for small problem sizes the most efficient algorithm is not necessarily the asymptotically best algorithm. Practice has demonstrated that, for small problem sizes, many asymptotically inferior algorithms behave better than the corresponding asymptotically superior algorithms. Furthermore, a multiplicity of algorithms perform considerably better on the *average* than predicted by the *worst-case* analysis.

2.1.2 The Optimality Notion

In this report, the performance of an algorithm is measured in terms of worst-case analysis (see previous section). Let Q be a problem for which we have a BSP algorithm that runs in time T_p on a p-processor machine. The processor-time product $C = pT_p$ represents the *cost* of the parallel algorithm. A BSP algorithm to solve Q will be called *optimal*, if the cost C required by the algorithm satisfies $C = \Theta(T_{seq})$, where T_{seq} denotes the time complexity of the fastest known sequential algorithm.[3] An optimal parallel algorithm will be called *cost-time optimal* if it can be shown that T_p cannot be improved (asymptotically) by any other optimal parallel algorithm.

Likewise, a BSP algorithm will be called *BSP-optimal*, if it can be shown that the cost C required by the algorithm cannot be improved by any other BSP algorithm. A BSP-optimal algorithm will be called *cost-time BSP-optimal* if it can be shown that T_p cannot be improved by any other optimal BSP algorithm. In this report we shall concentrate on *optimal* and *BSP-optimal* algorithms (unless otherwise indicated).

2.1.3 Arithmetic

It is not difficult to observe that, in computational geometry one must be able to deal with *real* numbers, not just integers, and, in practical situations, with proper approximations thereof. In the design of algorithms, however, one is allowed the conceptual abstraction to dispense with the round-off problem in the *approximate representation of real numbers* [137]. Later in this section we shall show how *exact arithmetic,* and *techniques for coping with degenerate cases* (see section 2.1.6) may relieve the programmer from the round-off problem and the task of providing a consistent treatment for every special case that may occur.

[3] Note that, the term T_{seq} appearing in the above definition is not necessarily the value of T_p for $p = 1$. Moreover, the cost C would be more naturally compared to the fastest sequential algorithm (rather than just the fastest known), but in general this quantity is difficult to determine [83].

2.1.4 Real Arithmetic and Basic Computational Operations

In the BSP model of parallel computation each *storage location* is capable of holding a single real number (*floating-point*),[4] and the following elementary operations are available at unit cost ($O(1)$ time):[5]

- The primitive *arithmetic operations* $(+, -, \times, /, modulo)$.
- The common *comparison operators* $(<, \leq, \geq, >, =, \neq)$.
- The local memory *read* and *write* operations (assignments).

Throughout this report, the term *basic computational operation* is employed to characterize such $O(1)$ time primitive operations. Additionally, the following functions may be considered as basic computational operations, since they can be obtained in constant time from the above primitive operations,

- log, ln,[6] n-th root, *exp*, $\lfloor x \rfloor$, $\lceil x \rceil$,[7] *random*.[8]

The assumptions made here are not completely unreasonable; this is because real numbers can be approximated very accurately, and therefore, the above operations and functions take constant time for all operands that can be stored in a single memory location of finite size. This model will be referred to as the *real BSP* model. It closely reflects the kinds of algorithms that are typically written in high-level algebraic languages, in which it is customary to consider variables of type *real* as having unlimited precision [137].

2.1.5 Exact Arithmetic

Even though, approximate (floating-point) arithmetic is a common and convenient substitution for real arithmetic, there are no simple techniques that guarantee the reliability of the resulting algorithm [92]. *Exact arithmetic* is another possible substitution [71]. The difficulty with exact arithmetic is that the required integer precision exceeds the native precision of all typical computers. Naive simulations of exact arithmetic - either modular or multiprecision integers - increase execution time dramatically over the floating-point alternative. In [71] it is shown that, the combination of two techniques, *tuned multiprecision integer arithmetic* and a *floating-point filter (based on interval analysis)*, can secure the effect of exact integer arithmetic at a cost close to that of floating point arithmetic.[9] The relative efficacy of these techniques depends upon the details of the problem instance in hand and the distribution of the input.

[4] Here the unrealistic assumption is made that real numbers can be stored in a computer.

[5] Once again, the assumption is unrealistic, because *any* operation on real numbers takes infinite time.

[6] Let $\log x$, $\ln x$ be the logarithms of x to base 2 and e respectively.

[7] Note that, the *floor* ($\lfloor x \rfloor$) of a number x is $x - (x \text{ modulo } 1)$, so that it is, too, a constant function. The same applies for the *ceiling* ($\lceil x \rceil$) of a number x.

[8] A random number generating function.

[9] In [71] experimental results demonstrated that the runtime never increased by more than 29%, with typical cases around $8 - 20\%$.

The first technique (tuned multiprecision integer arithmetic) improves the performance of naive multiprecision arithmetic by exploiting special constraints that computational geometry imposes on the use of multiprecision arithmetic. The following two aspects (constraints) are usually utilized [71],

(1) most arithmetic operations are of "medium length", thus the overhead per operation can be reduced to get good performance,

(2) while arithmetic expressions in computational geometry are fairly complicated, they do not change dynamically; therefore, it is reasonable to assume that optimization of an entire expression may be possible, rather than just an operation.

The second technique (floating-point filter) improves the performance of multiprecision arithmetic by reducing the use of exact arithmetic (using floating-point arithmetic as a filter to sort out easy cases).[10] Concluding, exact arithmetic techniques may offer the prospect of a significant improvement in the accuracy with which basic computational operations can be performed, at a modest cost.

2.1.6 Coping with Degeneracies

Degeneracies would never occur if geometric problems involved a finite number of geometric objects, each represented by a finite set of numbers, provided that there is no limitation on the precision of the numbers used. Unfortunately, in real-life computing, this is far from the norm; that is, the required number precision exceeds the native precision of all typical computers. Therefore, we are bound to deal with degenerate data [64].

Consider, for example, the elementary polygon inclusion problem - given a polygon P and a vector z, determine whether or not z is internal to P. Consider the horizontal line l passing through z. If l does not intersect P, it is a routine matter to verify that z is external. Thus, assume that l does intersect P. It is well known that z is internal if and only if the number of intersection points of l with the boundary of P to the left (or right) of z is odd [137]. It should be evident from figure 1 that, there are only two non-degenerate cases; however, there are four degenerate cases that have to be handled. As noted in [64], it is rather intricate to envision the peculiar structure of degenerate cases one faces in generalizing the polygon inclusion problem to higher dimensions.

Clearly, when implementing geometric algorithms, a substantial amount of effort is required to handle degeneracies. As problems become more involved degeneracies complicate, and, of course, the degree of robustness of the algorithm decreases. However, there are techniques to cope with degenerate cases. The basic idea behind all such techniques is to *perturb* the given input slightly so that an algorithm designed under the hypothesis of non-degeneracy can execute on arbitrary instances of input data. Note that, the perturbation should be small enough so that it does not modify the relative location of the non-degenerate objects [65]. Unfortunately,

[10] Floating-point filters are based on interval analysis; replace the evaluation of an expression by the evaluation of guaranteed upper and lower bounds on its value.

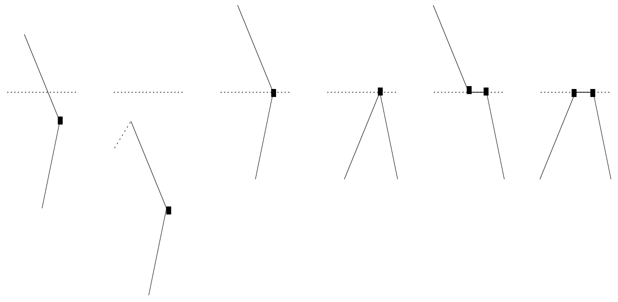

Figure 1: Polygon inclusion; non-degenerate cases (a)-(b), degenerate cases (c)-(f).

determining such a perturbation is fairly complicated and may require much higher precision than the original set of objects. For this reason, perturbations are realized only symbolically.

Even though, such techniques increase the time complexity of algorithms by a factor roughly proportional to the dimension of the geometric space [65], their main contribution is the introduction of a general technique that can be used to deal with degeneracies in a multiplicity of circumstances for arbitrary input instances in general position.[11] In the remainder of this report, we concentrate on non-degenerate geometric data inputs.

2.2 The Geometric Model

The objects considered in computational geometry are sets of vectors, lines/segments and hyperplanes in Euclidean space. Moreover, we assume that hyperplanes have an embedded Cartesian coordinate system of reference, so that each point is represented as a vector of the appropriate dimension.

2.2.1 General Concepts and Notation

This section has no pretence of providing formal definitions of the geometric notions used in this report, it merely introduces the adopted concepts and notations.

- By E^d we denote the *d-dimensional Euclidean space*, with metric $(\sum_{i=1}^{d} x_i^2)^{\frac{1}{2}}$.
- By $q = (x_1, ..., x_d)$ we denote a *point (vector)* q in E^d.
- The combination $aq_1 + (1-a)q_2$, for some real a, describes a *line* in E^d. The special case where $0 \leq a \leq 1$, characterizes a *straight line segment.* The straight line segment joining the two vectors q_1 and q_2 is denoted as $\overline{q_1 q_2}$.

[11] Since the computation of the perturbation requires the calculation of the sign of the determinant of an integer matrix, a promising way to decrease this overhead factor seems to be the introduction of special hardware that computes the sign of determinants for integer matrices [64].

- A straight line parallel to either the x-axis or y-axis is defined as an *isothetic straight line.*
- A domain D of vectors in E^d is *convex*, if for any pair of vectors in D, the straight line segment defined by these vectors belongs entirely in D [137].
- The *convex hull* of a set of vectors in E^d is defined as the smallest convex domain in E^d containing the entire set, i.e. given a set $S = (v_1, v_2, ..., v_n)$ of n vectors in E^d, the convex hull of S, denoted by $CH(S)$, is a convex polyhedron (polygon in E^2) whose corners are vectors of S such that every vector of S is either a corner of $CH(S)$ or lies inside $CH(S)$.
- A finite collection of straight line segments in E^2 such that every segment endpoint is shared by exactly two segments and no subset of edges has the same property, describes a *polygon.* A polygon is *simple* if there is no pair of segments sharing a common point (except endpoints). All simple non-convex polygons are described as *star-shaped.* The special case where the line segments are isothetic defines an *isothetic polygon.*
- In the same vein, a *polyhedron* in E^3 is defined by a finite collection of polygons such that every edge of a polygon is shared by exactly two polygons and no subset of polygons has the same property.
- A collection of vectors v (*vertex set*) and straight line segments e (*edge set*) describes a *graph.* A graph is *planar* if the set of edges and vertices composing the graph belong in E^2 without crossing. Let f denote the number of *faces* then, by *Euler's formula* [137], we have that $v+f-e = 2$. The special case where all its faces are triangles, defines a *triangulation.*

The restriction to Euclidean geometry is suggested by the fact that the vast majority of applications are formulated in Euclidean space, however, such a restriction is unessential for all of the applications to be considered in the remainder of this report [137].

2.3 The Computational Model

The term *bulk-synchronous* reflects the underlying position of the model between the two extremes: entirely synchronous systems (a global clock is utilized to achieve synchronization on every instruction), and fully asynchronous systems (synchronization is realized pairwise - pairs of processing units sending messages to each other - whenever communication is required). Processing units in the BSP model advance jointly through the program, with the required global (non-local) communication occurring between *supersteps.*

The *bulk-synchronous parallel (BSP)* computer as described in [27, 74, 115, 116, 124, 125, 165, 167] consists of three parts:

(1) a *set of p processing units*[12] numbered from $0...p-1$. Here, we shall assume that each processing unit consists of a sequential processor with a block of local memory,

[12] Throughout this report the terms *processing unit* and *processor* will be used interchangeably.

(2) a *communication network (router)* that can deliver messages in a point to point mode,

(3) a *mechanism* for efficient *barrier synchronization* of all, or a subset, of the processing units.[13]

The performance of any BSP computer can be precisely characterized by the following four parameters:

(1) p - the number of processing units.

(2) s - processor speed, i.e. number of *time steps* per second.[14]

(3) g - the ratio of the total throughput of the whole system in terms of basic computational operations (see section 2.1.3), to the throughput of the communication network in terms of words of information delivered. More formally, g is related to the time required to realize h-relations in a situation of heavy message traffic in gh time steps [116], for h larger than some h_0.[15] This constitutes a *uniform cost model* for non-local memory access.[16]

(4) L - the minimum elapsed time - measured in basic computational operations - between successive synchronization steps.

The special case $L = g = 1$ corresponds to the idealized PRAM, where local and non-local communication are not distinguished.

Computation on the BSP model proceeds in a succession of *supersteps*; a superstep may be conceived as a segment of a computation during which each processing unit performs a given thread (task) using data already available there locally before the start of the superstep. The thread may include (a) computation steps on values held locally at the start of the superstep, (b) message transmissions, (c) message receipts. At the end of each superstep, all processing units wait in a *global barrier synchronization* until the non-local data exchanges are completed before proceeding to the next superstep [125]. It should be noted that, although the model stresses global barrier mode synchronization, pairs of processing units may synchronize pairwise by sending messages to and from an agreed memory location. However, such message exchanges should respect the superstep rules [74].

The model as described in [165] has two variants depending on how synchronization is handled, that affect performance analysis to multiplicative constant factors. In the first variant, processing units are checked every L time units to determine whether the threads that are synchronized currently have all terminated and only then proceed to the next superstep. The second variant, permits completion of a superstep at any point after the first L time steps. For simplifying the analysis, we shall adopt the later variant.

[13]The idea of *barrier synchronization* is not a new concept (i.e. see [30, 66, 90, 102, 168]).

[14]A *time step* may be defined as the time required for performing a single basic operation.

[15]Note that, if $L \geq gh_0$ then every h-relation for $h < h_0$ will be charged as an h_0-relation [165].

[16]The assumption is made that the time required to transmit/receive a remote data item does not depend on its location. This conservative assumption simplifies the analysis and design of algorithms, however, may result in an overestimate of the actual cost of communication.

Architecture	*Degree*	*Diameter*	*Synchronization*	*Communication*
Ring	$\Theta(1)$	$\Theta(p)$	$L = O(p)$	$g = O(p)$
2D Array	$\Theta(1)$	$\Theta(\sqrt{p})$	$L = O(\sqrt{p})$	$g = O(\sqrt{p})$
Butterfly	$\Theta(1)$	$\Theta(\log p)$	$L = O(\log p)$	$g = O(\log p)$
Hypercube	$\Theta(\log p)$	$\Theta(\log p)$	$L = O(\log p)$	$g = O(1)$

Table 1: L and g values on different network architectures.

In particular, the complexity of a superstep S in a BSP algorithm is determined as follows. Each superstep is charged $x + \max\{gh, L\}$ basic time steps,[17] where x is the maximum number of basic computational operations executed by any processor during S, and h is the maximum number of messages transmitted or received by any processor. Note that, if h_t is the maximum number of messages transmitted by any processing unit during S, and h_r is the maximum number of messages received by any processing unit during S, then $h = \max\{h_t, h_r\}$, although $h = h_t + h_r$ is possible too [166]. Throughout this report the above analysis will be employed to predict the performance of BSP algorithms.

It is evident from the previous analysis that, the minimum size for a superstep is bounded by the frequency L with which barrier synchronization can be realized. On the other hand, the maximum size for a superstep is a property of the application, since by definition a superstep can accommodate as much work as can be done without a remote data exchange [125]. As it is noted in [74], any particular BSP machine may impose a lower bound on L (minimum size for a superstep) on a twofold basis:

(1) there is a minimum time requirement for doing synchronization,

(2) the theoretical bound on g may be realizable only for large enough L, so that the communication network can achieve the stated capacity.

Note that the particular values of L and g for any BSP machine[18] are likely to depend on p. In particular, the parameters L and g are non-decreasing functions of the number of processors, p, used by the program. As it is noted in [115], the values of L and g can be asymptotically estimated as indicated in table 1. [Note that, these estimates are based entirely on the *degree* and *diameter* characteristics of the underlying graph. In practical situations, considerations beyond degree and diameter would also have an important effect on the actual values of L and g, i.e. channel capacities, routing methods, etc. [115].] Reductions in the actual values of L and g may be realized by the efficient application of new optical technologies in the field of communications. For details on optical communication schemes one may refer to [14, 61, 76, 88, 134, 139, 165, 166, 173].

Concluding, we quote from [115],

[17] The original BSP model, charged each superstep $\max\{x, gh, L\}$ basic time steps; the difference between the two costs will not, in general, be significant [115].

[18] Any parallel computing system may be regarded as a BSP machine, and can be characterized by its BSP parameters p, s, g, and L [115].

> ... in the next few years we are likely to see a rapid convergence in the field of parallel computer systems. The various classes of parallel computer which currently exist (distributed memory architectures, shared memory multiprocessors, networks of workstations) will become more and more alike. This architectural convergence will enable parallel computing to become the normal method of computing, and will encourage the growth of a large and diverse parallel software industry similar to that which currently exists for sequential computing. It is now important that we establish a solid foundation to guide this process of convergence, and to enable architecture independent parallel software to be developed for the emerging range of scalable parallel systems. The BSP approach provides just such a foundation. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework which permits the performance of parallel and distributed systems to be analyzed and predicted in a precise way. Many exciting new research problems are emerging in this area as we try to understand the issues involved in the design and implementation of architecture independent programs for scalable parallel computing systems. We hope (and expect) that the research on these problems which is currently being pursued will enable the BSP approach to be further developed to the point where it can indeed provide a unifying framework for the design and programming of all kinds of parallel and distributed systems, analogous to that which the von Neumann model has provided for sequential systems over the last fifty years.

3 Geometric Searching

This section develops the basic tools and concepts of *geometric searching*,[19] that will be used in subsequent sections to tackle fairly formidable problems. Geometric searching problems [98, 111, 142] may be classified in two broad categories: *point location* and *range checking problems*. In point location problems [42, 103, 109, 160] we are given a partitioning of space P and an arbitrary point z, and we have to locate the relative position of z in P. On the other hand, in range checking problems [25, 28, 137, 155, 159] we are given a partitioning of space (range) R and an arbitrary set of points n, and we have to determine how many points lie in R.

In the simplest case, searching will be performed just once, in which case any preprocessing to speed up future queries would be wasteful. A one-time query will be referred to as *single-shot*. In many applications, however, numerous queries will be repeated on the same set of data. Such queries will be referred to as *repetitive mode queries* [137]. For this reason, algorithms for geometric searching are generally characterized by the following three attributes:

[19] Searching means to locate the appropriate record (or element) in a given collection of elements [105].

(1) *pre-processing time* - the time required in the construction of an appropriate search structure to speed up future queries,

(2) *space* - the storage required in the construction and representation of the search structure,[20]

(3) *query or search time* - the time required to respond to a single query.

In this section, we restrict our attention to the worst-case behaviour of these aspects. Note that, the trade-offs among pre-processing time, storage, and search time may lead to diverse algorithmic designs.

3.1 Point Location Problems

Point location problems could be suitably renamed *point inclusion problems*. Clearly, the complication of a point inclusion problem will essentially depend upon the dimension of the geometric space [137]. In the following sections some of the most fundamental two-dimensional point inclusion problems will be examined.

3.1.1 Polygon Inclusion

An elementary operation widely used in computational geometry is that of determining whether or not a point z is internal to a simple n-gon P. The basic idea behind the *polygon inclusion* problem is illustrated in figure 2. Consider the horizontal line l passing through z. If l does not intersect P, it is a routine matter to verify that z is external. Thus, assume that l does intersect P. It is well known that z is internal if and only if the number of intersection points of l with the boundary of P to the left (or right) of z is odd [137].

It is now straightforward to derive the following BSP algorithm. Assume that the polygon edges are initially distributed among the p processors equally but possibly arbitrarily (uniformly).

Algorithm 3.1

Input: *A polygon $P_{i=1\ldots n}$ of E^2 edges, and a E^2 point z.*
Output: *A boolean value indicating whether or not z is internal to P.*

Stage–1 Every processor (in parallel) computes the horizontal line l passing through z. [Note that, if the point z is not initially possessed by all processor, it can be distributed (broadcast) in $\log_t p \max\{tg, L\}$ time steps, without increasing the overall time complexity of the algorithm (see section A.1.1).]

Stage–2 Every processor (in parallel) counts the number of intersection points of l (to the left of z) with the portion of the boundary of P in its possession.

[20]In all algorithms to be presented in this report, $O(\frac{n}{p})$ space suffices (unless otherwise indicated).

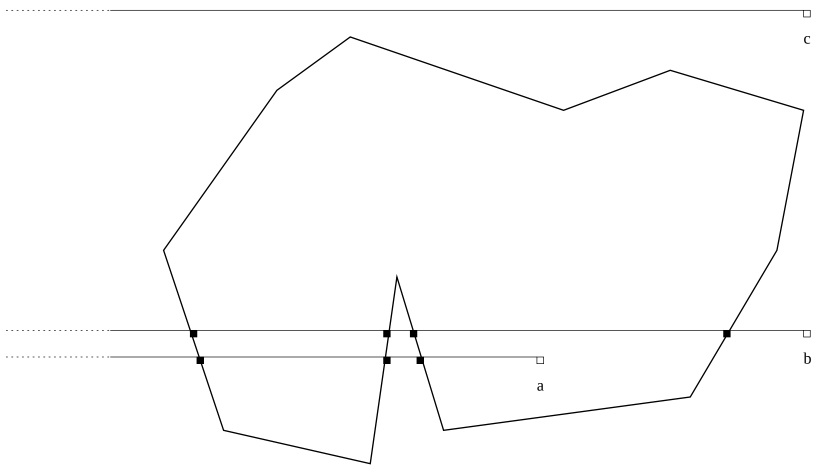

Figure 2: Polygon Inclusion.

Stage–3 Using a logical t-ary tree, the algorithm proceeds from the leaves to the root summing the number of intersection points.

Stage–4 A simple parity test is performed to determine whether or not z is internal - odd number of intersections indicates internal; otherwise external.

The correctness of the algorithm follows directly from the discussion preceding it. The time complexity of the algorithm can be estimated as follows.

Theorem 3.1 *There exists a BSP algorithm that, given a simple n-gon P and a point z, can determine whether or not z is internal to P in*

$$O(\frac{n}{p} + \log_t p \max\{tg, L\})$$

time (without pre-processing).

Proof: The parity test (stage 4) and the determination of the horizontal line l (stage 1) take constant time. The local counting of the number of intersection points requires $O(\frac{n}{p})$ time steps. Finally, the communication complexity (execution of t-ary tree) takes time $O(\log_t p \max\{tg, L\})$ (see section A.2.1). We thus obtain the above theorem. ■

The following note deals with the notion of optimality (note that the sequential complexity of the polygon inclusion problem is $O(n)$ [137]).

Corollary 3.1 *The polygon inclusion problem can be solved in optimal*

$$O(\frac{n}{p})$$

time, on the BSP model, for any integers t, n, p, L, and g such that $t = O(\frac{n}{pg\log p}\log\frac{n}{pg\log p})$, $L = O(\frac{n}{p\log_t p})$, *and* $n = \Omega(ptg\log_t p)$.

Proof: Similar to that of lemma A.1. ■

3.2 Range Searching Problems

Range searching problems may be considered as dual, in some sense, of the point location problems we have discussed so far. However, in the range searching context one deals exclusively with range queries. In the following section an algorithm for the two-dimensional dominance counting problem is developed that can be employed to solve some of the most fundamental range checking problems.

3.2.1 Dominance Counting

In this section, we consider the following version of the *dominance counting problem* [119, 156]. Given a set A of m points and a set B of k points in the plane, determine for each point z in A the number of points in B that are dominated by z. The *dominance* is a relation on a set S of n vectors as follows.

Definition 3.1 *Vector u dominates vector v if $x_i(u) \leq x_i(v)$ for all $i = 1, 2, ..., d$.*

It is clear that the brute force algorithm[21] results in $O(\frac{mk}{p})$ time steps. Undoubtedly, the result is non-optimal since an optimal $O(n\log n)$ sequential algorithm exists [137], where $n = m + k$. In the remainder of this section, we explore the possibility that the brute force algorithm may be doing unnecessary work.

Assume that, $D(z, B)$ is the number of points in B dominated by z. We start with the following lemma.[22]

Lemma 3.1 *Let V and H be a vertical and a horizontal slab of B, delimited from left and below by the lines v and h respectively. For every point z in A that belongs to the intersection I of V and H, $D(z,B) = D(z,V) + D(z,H) - D(z,I)$ + (the number of points in B that are both below h and to the left of v).*

Proof: Immediate. ■

The algorithm that follows derives from the *2D-Weighted Dominance Counting* algorithm of [55] and the above lemma. The assumption is made that the input is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

[21] Examine for every point in A each of the points in B to see whether it satisfies the dominance property.

[22] An analogous result appears in [55].

Algorithm 3.2

Input: *Two arrays* $A_{i=1\ldots m}$ *and* $B_{i=1\ldots k}$ *of* E^2 *vectors, such that* $n = m + k$.
Output: *An array* $C_{i=1\ldots m}$ *of integers, such that* $C_i = t$, *iff* A_i *dominates exactly* t *points of* B.

Stage–1 The bucketing algorithm of section B.2 is used to route the points in A and B to the appropriate bucket (processor), using as key the x-coordinate, which yields set V_i on processor p_i. [Notice that the resulting buckets are of size $< (1 + \frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–2 Each processor p_i solves (sequentially) the dominance counting problem for the points in V_i. Additionally, each processor computes the minimum and the maximum of the points in its possession.[23]

Stage–3 A global broadcast operation is initiated, where processor p_i sends line l_i, the vertical line passing through the point with maximum x-coordinate, to all other processors.

Stage–4 The bucketing algorithm of section B.2 is used to route the points in A and B to the appropriate bucket (processor), using as key the y-coordinate, which yields set H_j on processor p_j. [Notice that the resulting buckets are of size $< (1 + \frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–5 Each processor p_j solves (sequentially) the dominance counting problem for the points in H_j.

Stage–6 Each processor $p_j$determines the intersections of H_j and V_i, for all i such that $0 \leq i < p$.

Stage–7 Each processor p_j solves the dominance counting problem for the points in the intersection of H_j and V_i, for all i such that $0 \leq i < p$.

Stage–8 Each processor p_j computes s_{ij}, the number of points z in H_j that belong to B and $z_x < l_i$, for all i such that $0 \leq i < p$.

Stage–9 A global broadcast operation is initiated, where processor p_j sends s_{ij} to processor p_{i+1}, for all i such that $0 \leq i < p - 1$.

Stage–10 Each processor solves the dominance counting problem for the points in its possession using the formula from lemma 3.1.

The correctness of the algorithm follows directly from lemma 3.1.

[23]The minimum and the maximum may result as by-products of the dominance counting problem solution.

Theorem 3.2 *There exists a BSP algorithm that, given a set A of m points and a set B of l points in the plane, such that $n = m + k$, can determine for each point z in A the number of points in B that are dominated by z in*

$$O(\quad \begin{array}{l} p \ln^2 n \log (p \ln^2 n) + \\ \frac{n}{p} \log p + \\ \max \{gp \ln^2 n, L\} + \\ \log_t p \max \{tgp, L\} + \\ \max \{g\kappa \frac{n}{p}, L\} + \\ \kappa \frac{n}{p} \log (\kappa \frac{n}{p})) \end{array}$$

time - with probability at least $1 - O(\frac{1}{n})$ - where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The bucketing algorithm in stages 1 and 4 requires $O(p \ln^2 n \log (p \ln^2 n) + \frac{n}{p} \log p + \max \{gp \ln^2 n, L\} + \log_t p \max \{tgp, L\} + \max \{g\kappa \frac{n}{p}, L\})$ time (see theorem B.3). The sequential solution of the dominance counting problem in stages 2, 5, and 7 can be realized in time $O(\kappa \frac{n}{p} \log (\kappa \frac{n}{p}))$. The computation in stage 6 can be performed in $O(\kappa p \log p)$ time. Stage 8 can be realized in $(O(\kappa \frac{n}{p} \log p)$ time, using binary search. The communication complexity in stages 3 and 9 is bounded above by $O(\max \{gp, L\})$. Finally, the dominance counting computation in stage 10 can be realized in $O(\kappa \frac{n}{p})$ time steps. We thus obtain the above theorem. ∎

The following note deals with the notion of optimality.

Corollary 3.2 *The dominance counting problem can be solved in optimal*

$$O(\frac{n \log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n \log n}{p \log p})$, and $g = O(\log n)$.

Proof: Let in theorem 3.2 $t = 2$. ∎

Note that, the probabilistic feature of algorithm 3.2 can be eliminated if the bucketing algorithm of section B.2 is replaced by a similar deterministic bucketing algorithm (i.e., see algorithm B.4).

3.2.2 Rectangle Inclusion

A range checking problem that can be handled efficiently using dominance counting is the *rectangle inclusion problem.* Given a set A of m points in the plane, and a set P of r rectangles whose sides are parallel to the coordinate axes, determine, for each rectangle R in P, how many points of A lie in R. It is clear that the brute force algorithm results in $O(\frac{rm}{p})$ time,[24] which is unacceptable for most real life applications. In the rest of this section an optimal $O(\frac{n \log n}{p})$ algorithm is developed, where $n = m + 4r$. Note that, the lower bound for sequential rectangle inclusion is $\Omega(n \log n)$ [137].

Assume that the sets P and A are initially distributed among the p processing units equally but possibly arbitrarily (uniformly). We start with the following lemma.

Lemma 3.2 *The rectangle inclusion problem can be computed in $O(\frac{r}{p})$ time, on the BSP model, once the dominance counting problem has been solved, where r is the number of rectangles.*

Proof: Let R be a rectangle with vertices (z_1, z_2, z_3, z_4) ordered counterclockwise starting from the upper right corner. The number of points contained in rectangle R is given by,

$$D(z_1) - D(z_2) - D(z_4) + D(z_3), \tag{1}$$

where $D(z)$ denotes the number of points in A dominated by vertex z. This follows directly from the combinatorial principle of inclusion-exclusion [126]. Hence, such a number can be computed in $O(1)$ sequential time for each rectangle. Since the number of rectangles is r, the rectangle inclusion problem can be computed in $O(\frac{r}{p})$ time, once the dominance counting problem has been solved. ■

Now we can proceed with the rectangle inclusion algorithm.

Algorithm 3.3

Input: *An array $A_{i=1...m}$ of E^2 vectors, and an array $P_{i=1...r}$ of E^2 rectangles, such that $n = m + 4r$.*
Output: *An array $C_{i=1...r}$ of integers, such that $C_i = t$, iff P_i contains exactly t points of A.*

> **Stage–1** Solve the dominance counting problem with respect to the pair A and P.
>
> **Stage–2** Each processor computes for each rectangle R in its possession the number of points of A that lie in R, using the formula from lemma 3.2.

The correctness of the algorithm follows directly from lemma 3.2.

Theorem 3.3 *There exists a BSP algorithm that, given a set A of m points in the plane, and a set P of r rectangles whose sides are parallel to the coordinate axes,*

[24]Examine (for every rectangle) each of the m points to see whether it satisfies the inclusion property.

such that $n = m + 4r$, can determine, for each rectangle R in P, how many points of A lie in R, in

$$
\begin{aligned}
O(\quad & p\ln^2 n \log(p\ln^2 n) + \\
& \frac{n}{p}\log p + \\
& \max\{gp\ln^2 n, L\} + \\
& \log_t p \max\{tgp, L\} + \\
& \max\{g\kappa\frac{n}{p}, L\} + \\
& \kappa\frac{n}{p}\log(\kappa\frac{n}{p}))
\end{aligned}
$$

time - with probability at least $1 - O(\frac{1}{n})$ - where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The local computation in stage 2 takes time $O(\frac{r}{p})$ (see lemma 3.2) and is undoubtedly dominated by the dominance counting time complexity of stage 1 (see theorem 3.2). We thus obtain the above result. ■

The following corollary deals with the notion of optimality.

Corollary 3.3 *The rectangle inclusion problem can be solved in optimal*

$$O(\frac{n\log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n\log n}{p\log p})$, and $g = O(\log n)$.

Proof: Let in theorem 3.3 $t = 2$. ■

Note that, the probabilistic feature of algorithm 3.3 can be eliminated if the dominance counting algorithm of section 3.2.1 is replaced by a similar deterministic algorithm (i.e., see algorithm B.4).

4 Intersection Problems

Much of the impetus for studying intersection problems arises from the simple fact that more than one objects cannot occupy the same place at the same time. Because two objects intersect only if one contains a point of the other it is evident that intersection problems may involve testing for inclusion [137]. Hence, we may consider intersection problems [81, 119] to be inherent extensions of point inclusion problems treated in section 3.1. The importance of efficient intersection algorithms is apparent in many applications. Thus, for example, in pattern recognition [120] it is important to determine whether different regions in space have common subranges, in VLSI design [87] it is crucial to avoid crossing wires and overlapping components, and in computer graphics [137] it is necessary for hidden line/surface removal.

4.1 Intersection of Horizontal and Vertical Line Segments

One of the major problems in computational geometry is that of determining the number of *intersections among horizontal and vertical line segments.* In particular, given a set $S = H \cup V$ of m horizontal segments H and k vertical segments V, the intersection problem is to determine, for each segment s in S, the number of segments in S that intersect s. The applications of an efficient solution of the pre-mentioned problem range from computer graphics to VLSI design.

Obviously the brute force algorithm results in $O(\frac{mk}{p})$ time,[25] which is unacceptable for most of the real life applications. In the rest of this section an optimal $O(\frac{n \log n}{p})$ algorithm is developed, where $n = 2m + 2k$. Note that, the lower bound for the sequential algorithm is $\Omega(n \log n)$ [137].

Assume that the sets H and V are initially distributed among the p processing units equally but possibly arbitrarily (uniformly). We start with the following lemma.

Lemma 4.1 *The horizontal and vertical segments intersection problem can be computed in $O(\frac{n}{p})$ time, on the BSP model, once the dominance counting problem has been solved.*

Proof: Let S_l and S_r be, respectively, the sets of the left and right endpoints of all the horizontal segments. The number of segments intersecting any vertical segment $\overline{pq}$, where $p_y > q_y$, is given by,

$$D_l(p) - D_r(p) - D_l(q) + D_r(q), \tag{2}$$

where $D_l(z)$ and $D_r(z)$ denote, respectively, the number of left and right horizontal segment endpoints dominated by point z. It is straightforward to verify the above claim, if we note that $D_l(z) - D_r(z)$ is the number of horizontal segments below z that cross the vertical line through z. Hence, such a number can be computed in $O(1)$ sequential time for each vertical segment, once the dominance counting problem has been solved. We thus obtain the above lemma. ■

Now we can proceed with the horizontal and vertical segments intersection algorithm.

Algorithm 4.1

Input: *Two arrays $H_{l=1\ldots m}$, $H_{r=1\ldots m}$ of E^2 vectors (corresponding, respectively, to the left and right endpoints of the horizontal segments), and an array $V_{i=1..k}$ of E^2 vertical segments, such that $n = 2m + 2k$.*
Output: *An array $C_{i=1\ldots l}$ of integers, such that $C_i = t$, iff V_i is intersected by exactly t segments $\in S$.*

[25] Determine for each vertical segment whether or not it intersects any of the k horizontal segments. Repeat the same process for the horizontal segments.

Stage–1 Solve the dominance counting problem with respect to the pairs H_l, H_r and V.

Stage–2 Each processor computes for each vertical segment s in its possession the number of horizontal segments that intersect s, using the formula from lemma 4.1.

The correctness of the algorithm follows directly from lemma 4.1. In a similar fashion, we can find, for each horizontal segment the number of vertical segments that intersect it.

Theorem 4.1 *There exists a BSP algorithm that, given a set S of n horizontal and vertical segments in the plane, can determine, for each segment s, the number of segments in S that intersect s, in*

$$\begin{aligned} O(\quad & p \ln^2 n \log (p \ln^2 n) + \\ & \frac{n}{p} \log p + \\ & \max \{gp \ln^2 n, L\} + \\ & \log_t p \max \{tgp, L\} + \\ & \max \{g\kappa \frac{n}{p}, L\} + \\ & \kappa \frac{n}{p} \log (\kappa \frac{n}{p})) \end{aligned}$$

time - with probability at least $1 - O(\frac{1}{n})$ *- where* $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The local computation in stage 2 takes time $O(\frac{n}{p})$ (see lemma 4.1) and is undoubtedly dominated by the dominance counting time complexity of stage 1 (see theorem 3.2). We thus obtain the above result. ■

The following note deals with the notion of optimality.

Corollary 4.1 *The horizontal and vertical segments intersection problem can be solved in optimal*

$$O(\frac{n \log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n \log n}{p \log p})$, and $g = O(\log n)$.

Proof: Let in theorem 4.1 $t = 2$. ■

Note that, the probabilistic feature of algorithm 4.1 can be eliminated if the dominance counting algorithm of section 3.2.1 is replaced by a similar deterministic algorithm (i.e., see algorithm B.4).

5 Convex Hulls

The problem of computing the *convex hull* of a point-set [8, 9, 34, 36, 37, 49, 80, 121, 122, 123, 170] is not only fundamental to many practical applications, but is also a mean for the solution of apparently unrelated issues arising in computational geometry. The importance of efficient convex hull algorithms is apparent in many applications; for example, in pattern recognition [163], in image processing [150] and in statistics [154].

The *convex hull* $CH(S)$ of a set S of points in E^d is defined as the smallest convex domain in E^d containing the entire set (see section 2.2). Unfortunately, this simple definition is not of a constructive nature. Thus, appropriate notions must be identified that are conductive to algorithm development.

5.1 Problem Statement and Lower Bounds

We begin by exposing the two most favoured statements of the convex hull problem [137].

Convex Hull Given a set S of n points in E^d, construct its convex hull, that is, the complete description of the boundary of $CH(S)$.

Extreme Points Given a set S of n points in E^d, identify those points that are vertices of $CH(S)$.

Clearly, the first problem is asymptotically at least as hard as the second. In the remainder of this report we shall concentrate on the first version of the convex hull problem.

The following is shown in [137, 176].

Lemma 5.1 *Sorting is linear-time transformable to the planar convex hull problem; therefore, finding the ordered convex hull of n points in the plane requires $\Omega(n \log n)$ time steps.*

Proof: See [137]. ■

The transformation required involves only arithmetic operations, therefore, the above lemma holds in the BSP model of parallel computation.

We shall now begin the description of the convex hull algorithms.

5.2 Convex Hulls in the Plane

The computation of the planar convex hull has been studied extensively in past two decades. This section will consider two BSP algorithms for solving the two-dimensional convex hull problem [8, 9, 12, 37, 63, 93, 127]. The output of these algorithms will be a list of all the vertices of the convex hull in counterclockwise order around its boundary.

5.2.1 An Optimal Expected-Time Algorithm

In this section we show how the convex hull vertices of a vector set can be identified in optimal expected-time, on the BSP model. The algorithm to be presented computes the convex hull of a finite set of points by a technique known as *gift wrapping*. The basic idea behind this method is stated in the following lemma.

Lemma 5.2 *A line segment $\overline{pq}$ defined by two points p and q is an edge of the convex hull if and only if all other points of the set lie on $\overline{pq}$ or to one side of it.*

Proof: The correctness of the above lemma follows directly from the convex hull definition (see section 2.2). ■

Clearly, there are $O(n^2)$ lines determined by all pairs of n points. For each of these lines we may examine the remaining $n-2$ points and apply the above lemma to determine in $O(n)$ sequential time whether or not the segment is a convex hull edge. This technique results in $O(n^3)$ sequential time, which is unacceptable for most real life applications. In [97], Jarvis observed that this algorithm can be improved if we note that once we have found that $\overline{pq}$ is a convex hull edge, then some other edge must exist with q (or p) as an endpoint.

Using this fact we can proceed with the following convex hull algorithm. Assume that the set S of n points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 5.1

Input: *An array $S_{i=1\dots n}$ of E^2 vectors.*
Output: *An ordered array $CH_{i=1..n}$, containing the convex hull $CH(S)$ of S.*

Stage–1 The algorithm of section A.2.2 is used to find the point in S with minimum x-coordinate. [Note that, this point belongs to the convex hull of S.]

Stage–2 Each processor selects, from the points in its possession, the point with minimum polar angle, with respect to the previously determined point as origin. [Note that, the smallest angle can be found with arithmetics and comparisons alone, without actually computing any polar angles, i.e. compute the cross product of two points p and q; if the cross product is positive then p is clockwise from q with respect to the origin, otherwise p is counterclockwise from q.]

Stage–3 A reduce operation is initiated (on the polar angles), using the binary operator *minimum*. [Note that, the resulting polar angle corresponds to the next point on the convex hull perimeter (see lemma 5.2).]

Stage–4 Stages 2 and 3 are repeated until the algorithm returns to the point determined in stage 1.

The correctness of the algorithm follows directly from the discussion preceding it.

Theorem 5.1 *There exists a BSP algorithm that, given a set S of n points in the plane, can construct the convex hull $CH(S)$ of S in*

$$O(h(\frac{n}{p} + \log_t p \max\{tg, L\}))$$

time, where h is the number of vertices of $CH(S)$.

Proof: The computation in stage 1 takes time $O(\frac{n}{p} + \log_t p \max\{tg, L\})$ (see section A.2.2). The time required for stage 2 is obviously $O(\frac{n}{p})$. Finally, the time complexity of the reduce operation (stage 3) is $O(\log_t p \max\{tg, L\})$ (see section A.2.1). We thus obtain the above result. ■

Since all n points of a set may lie on its convex hull ($h = n$), the worst-case running time of algorithm 5.1 is $O(\frac{n^2}{p})$, which is inferior to the optimal $O(\frac{n \log n}{p})$ bound, however, if it is known in advance that h is small (i.e. $o(\log n)$) then the algorithm defeats the optimal lower bound! For example, it is known that, $h = O(\log n)$ holds for multivariate uniform distributions, $h = O(\sqrt{\log n})$ holds for multivariate normal distributions, and finally $h = O(1)$ holds for multivariate t-distributions drawn from a unit d-cube [60]. This observation is quite important and will be further treated in section 5.3.

From theorem 5.1 it follows directly that,

Corollary 5.1 *The convex hull problem can be solved in optimal expected time (i.e. $h = O(\log n)$)*

$$O(\frac{n \log n}{p}),$$

on the BSP model, for any integers n, p, L, and g such that $L = O(\frac{n}{p \log_t p})$, and $g = O(\frac{n}{tp \log_t p})$.

Proof: Immediate. ■

In the section that follows an optimal worst-case algorithm is presented. It should be noted that, even though the worst-case asymptotic behaviour of the algorithm presented in this section is inferior to that of section 5.2.2, it is expected that, on the average, algorithm 5.1 will outperform the algorithm of section 5.2.2.

[To demonstrate the practical relevance of our scalable BSP algorithms, we implemented the above convex hull algorithm on a Silicon Graphics Power Challenge shared memory system (consisting of four 64-bit Streaming Superscalar TFP processors) and an IBM 9076 close-coupled distributed memory system (consisting of eight IBM RS/6000 model 580 processors). The results of our experiments can be found in appendix C section C.3.]

5.2.2 An Optimal Worst-Case Algorithm

In this section we show how the convex hull of a set of points can be constructed in optimal worst-case time, on the BSP model. The basic idea behind the algorithm of this section is presented in the following two lemmas.

Lemma 5.3 *The convex hull $CH(S)$ resulting from the union*[26] *of $CH(S_1) \cup CH(S_2) \cup ... \cup CH(S_i)$ is equal to the convex hull of the union of $CH(S_1 \cup S_2 \cup ... \cup S_i)$, i.e. the convex hull of S is invariant under permutations of $(S_1, S_2, ..., S_i)$, and possesses the inclusion property.*

Proof: The correctness of the above lemma follows directly from the definition of the convex hull (see section 2.2). ■

Lemma 5.4 *Given two convex hulls $CH(S_1)$ and $CH(S_2)$ of size $O(n)$ each, such that $p_x < q_x$ for all $p \in S_1$ and $q \in S_2$, a single processor can compute the upper/lower tangent of $CH(S_1)$ and $CH(S_2)$ in $O(\log n)$ time steps.*

Proof: A detailed proof of the above lemma is given in [129]. ■

The main algorithmic idea is similar to the partition method in [3, 55]. Since the algorithm to be presented employs a sequential convex hull algorithm, it is assumed that an optimal $O(n \log n)$ sequential convex hull algorithm exists [137]. Additionally, assume that the set S of n points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 5.2

Input: *An array $S_{i=1...n}$ of E^2 vectors.*
Output: *An ordered array $CH_{i=1...n}$, containing the convex hull $CH(S)$ of S.*

Stage–1 The bucketing algorithm of section B.2 is used to route the points in S to the appropriate bucket (processor), using as key the x-coordinate, which yields set S_i on processor p_i. [Notice that the resulting buckets are of size $< (1 + \frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–2 Each processor p_i solves (sequentially) the convex hull problem for the points in S_i.

Stage–3 A global broadcast operation is initiated, where every processor p_i sends the median element (point) of S_i (the array holding the convex hull chain), its successor and its predecessor, to all other processors.

Stage–4 All pairwise upper (and lower) tangents are computed using a procedure similar to that of [129]: each processor performs $2(p-1)$

[26] An extended notion of the term *union* is used to accommodate convex hulls.

binary searches to determine the $2(p-1)$ pairwise tangents. [Note that, since the binary search procedure of [129] requires (at each step of the search procedure) a point, its successor and its predecessor, a global broadcast operation is initiated at each step of the binary search procedure, where every processor sends $2(p-1)$ points, their successors and their predecessors, to the appropriate processors.]

Stage–5 Each processor computes its contribution (possibly empty) to the convex hull of S by selecting the sequence of points on the upper chain (the lower chain can be computed in a similar fashion) between the leftmost point of tangency v_r of a right tangent e_r and the rightmost point of tangency v_l of a left tangent e_l [80]. As shown in figure 3 if the angle formed by e_l and e_r is less than π, none of the points between v_r and v_l is on the convex hull; otherwise, all the points between v_r and v_l are on the convex hull.[27]

Stage–6 Processors p_0 and p_{p-1} broadcast, respectively, the points with minimum and maximum x-coordinates to all other processors. Note that, the line passing through these two points defines the upper and lower chains of the convex hull.

Stage–7 Each processor determines for the points in its possession, whether they belong to the upper or lower chain. Note that, the points on both chains are sorted in ascending x-coordinate.

Stage–8 The number of points that fall into each of the two chains are counted, and a multiple all-prefix operation is then initiated on the number of points per chain and per processor to find the processor destination of each point.

Stage–9 Each processor knows where to route every point in its possession, and performs this routing.

The correctness of the algorithm follows directly from lemmas 5.3 and 5.4.

Theorem 5.2 *There exists a BSP algorithm that, given a set S of n points in the plane, can construct the convex hull $CH(S)$ of S in*

$$
\begin{aligned}
O(\quad & p \ln^2 n \log (p \ln^2 n) + \\
& \frac{n}{p} \log p + \\
& \kappa \frac{n}{p} \log (\kappa \frac{n}{p}) + \\
& \max \{gp \ln^2 n, L\} + \\
& \log_t p \max \{tgp, L\} +
\end{aligned}
$$

[27]Note that, the non-ordered convex hull is available at this stage; stages 8 to 11 are performed to obtain the ordered convex hull.

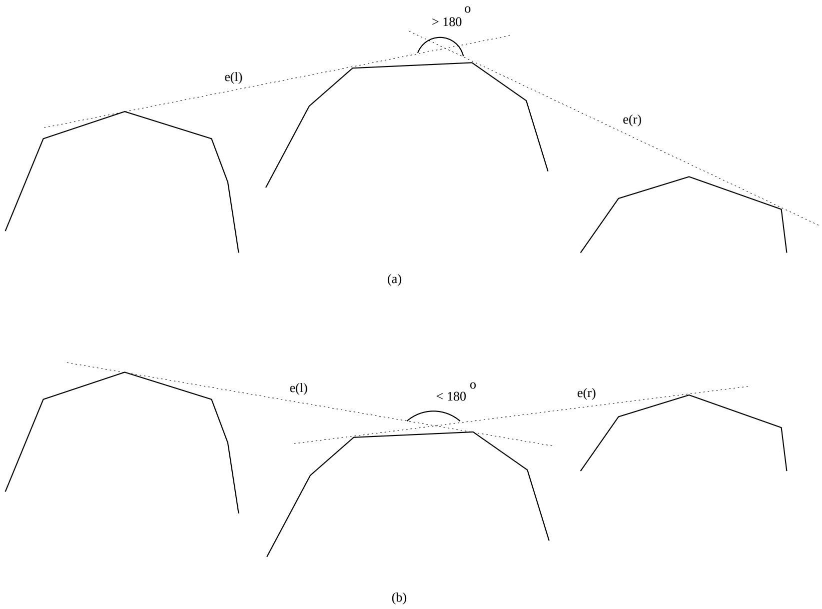

Figure 3: Two dimensional convex hull merging: If the angle formed by e_l and e_r is less than π, none of the points between the tangency points of e_l and e_r is on the convex hull; otherwise, all the points are on the convex hull.

$$\max\{g\kappa\frac{n}{p}, L\} +$$
$$\log(\kappa\frac{n}{p})\max\{pg, L\})$$

time - with probability at least $1 - O(\frac{1}{n})$ *- where* $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The bucketing algorithm in stage 1 requires $O(p\ln^2 n\log(p\ln^2 n) + \frac{n}{p}\log p + \max\{gp\ln^2 n, L\} + \log_t p\max\{tgp, L\} + \max\{g\kappa\frac{n}{p}, L\})$ time steps (see theorem B.3). The sequential solution of the convex hull problem in stage 2 can be computed in time $O(\kappa\frac{n}{p}\log(\kappa\frac{n}{p}))$. Stages 3 and 4 can be realized in time $O(\log(\kappa\frac{n}{p})\max\{pg, L\})$. The local computation of stage 5 takes time $O(\kappa\frac{n}{p})$. Finally, stages 6, 7, 8, and 9 are similar to those of the bucketing algorithm of stage 1, and, therefore, they do not contribute to the overall time complexity of the algorithm. We thus obtain the above theorem. ■

Accordingly,

Corollary 5.2 *The two-dimensional convex hull problem can be solved in optimal*

$$O(\frac{n\log n}{p})$$

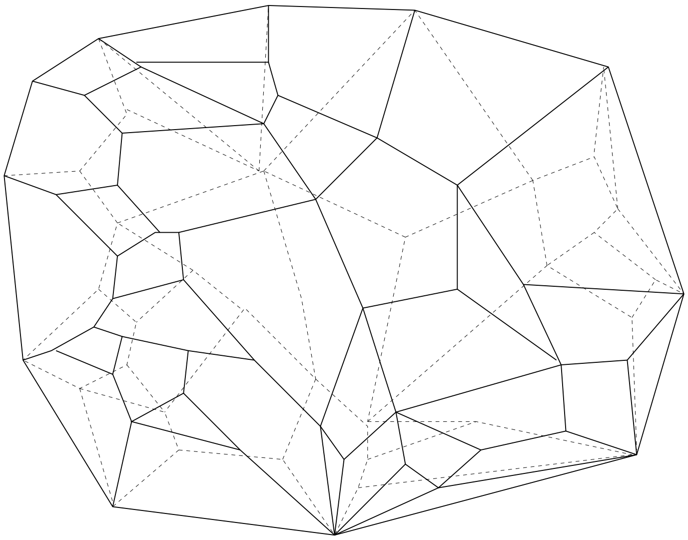

Figure 4: Three-dimensional convex hull.

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\log n})$, $L = O(\frac{n \log n}{p \log p})$, and $g = O(\log n)$.

Proof: Let in theorem 5.2.2 $t = 2$. ■

5.3 Convex Hulls in Higher Dimensions

In this section we present a high-probability optimal $O(k\frac{n}{p}\log\frac{n}{p})$ time BSP algorithm for computing the convex hull of n points in E^3, for some constant k, under the assumption that the points are drawn independently from any identically distributed continuous function (see figure 4 for a three-dimensional convex hull). Examples of distributions that satisfy the above condition include the multivariate normal distribution and multivariate uniform distribution drawn from a unit d-cube [23]. Furthermore, the algorithm may be adopted to higher dimensions with no modifications at all! Note that, all adequately documented deterministic parallel algorithms proposed for the general three-dimensional convex hull problem use time at least $O(\frac{n \log^2 n}{p})$.

In the following section we introduce the mathematical framework required for the analysis of the convex hull algorithm of section 5.3.2.

5.3.1 Preliminaries and Mathematical Notions

Let S be a set of independently distributed random vectors[28] $(v_1, v_2, ..., v_n)$ in E^d . While it is true that, by *Markov's inequality* [133], [29]

$$Prob[T \geq cE(T)] \leq \frac{1}{c}, \text{ for all } c > 0, \tag{3}$$

this bound is too crude to be of any realistic interest. An alternative to Markov's inequality is offered by *Chebyshev's inequality* [133],

$$Prob[(T - E(T)) \geq c] \leq \frac{Var(T)}{c^2}, \text{ for all } c > 0. \tag{4}$$

In this section, an extension of the above inequality will be used [133],

$$Prob[T \geq c] \leq \frac{E(T^2)}{c^2}, \text{ for all } c > 0, \tag{5}$$

where $E(T^2) = Var(T) + E(T)^2$ [133]. This requires usually a more intricate analysis than Markov's inequality, however, the probability bound obtained is as strong as high probability bounds whenever $(Var(T) + E(T)^2)/c^2$ has a zero limit as the problem size tends to infinity.

The problem of finding the average number of maximal vectors in a set of n E^d-valued vectors has been studied extensively in the past. The following general result, for the expected number of maxima in E^d, is due to [26].

Theorem 5.3 *The average number of maxima in a set S of n independently distributed random vectors in d-space is bounded above by $O((\ln n)^{d-1})$ for any fixed d.*

Proof: A detailed proof of the above result is given in [26]. ■

Additional results can be found in section 5.3.2.

Despite the elegancy of some of the techniques used to establish the pre-mentioned result, similar techniques could not be employed to obtain an estimate for the variance of the number of maxima in d-space. In the remainder of this section, a different approach is used to calculate the exact value of the variance for the two-dimensional case, and then the general result of [58] for the variance in d-space is presented.

To compute the variance of the expected number of maxima, of a two-dimensional vector-set, we shall consider, without loss of generality, the set sorted by x-coordinate. It is clear that, the i-th vector is a maximal if and only if its y-coordinate is maximal in the set of the i-th through the n-th vectors. Since, the assumption is that all $n!$ permutations are equally likely, the probability P_{ni} that the expected number of maxima $E(|MAX(S)|)$ equals i will be,

[28] The assumption of independently distributed random vectors secures that all $(n!)^d$ relative orderings are equally probable.

[29] $E(X)$ and $Var(X)$ are used, respectively, for the expected value and variance of the random variable X.

$$P_{ni} = \frac{U_{(i)}^{(n)}}{n!}, \tag{6}$$

where $U_{(i)}^{(n)}$ is the number of permutations of n points for which $E(|MAX(S)|) = i$. Furthermore, the expected number of maxima is by definition [133],

$$E(|MAX(S)|) = \sum_i iP_{ni}. \tag{7}$$

Similarly, the variance $Var(|MAX(S)|)$ is defined to be [133],

$$Var(|MAX(S)|) = \sum_i i^2 P_{ni} - E(|MAX(S)|)^2. \tag{8}$$

Now we can determine the behaviour of $E(|MAX(S))$ and $Var(|MAX(S)|)$ by calculating P_{ni}. If the y-coordinate of the first vector equals n, then the value of $E(|MAX(S)|)$ is one higher than the value obtained on the remaining vectors (as before, we consider the set sorted by x-coordinate in ascending order; furthermore, we assume that all numbers have been translated to integers from 1 to n). Otherwise, the value of $E(|MAX(S)|)$ is exactly the same as its value on the remaining vectors [104]. Therefore,

$$P_{ni} = (\frac{1}{n})P_{(n-1)(i-1)} + (\frac{n-1}{n})P_{(n-1)i}, \text{ for all } n > 1, i > 0, \tag{9}$$

where $P_{ni} = 0$ for all $i < 0$. Define now the probability generating function,

$$g_n(z) = \sum_i P_{ni} z^i, \tag{10}$$

which is a polynomial in z of degree n. By using differentiation we have,

$$g_n'(z) = \sum_i iP_{ni} z^{i-1}. \tag{11}$$

Combining the above result with equation 7 gives,

$$E(|MAX(S)|) = \sum_i iP_{ni} = g_n'(1). \tag{12}$$

Similarly, by applying differentiation once again we obtain,

$$Var(|MAX(S)|) = \sum_i i^2 P_{ni} - E(|MAX(S)|)^2 = g_n''(1) + g_n'(1) - g_n'(1)^2. \tag{13}$$

Combining equations 9 and 10 we have,

$$g_n(z) = (\frac{z}{n})g_{n-1}(z) + (\frac{n-1}{n})g_{n-1}(z) = (\frac{z+n-1}{n})g_{n-1}(z). \tag{14}$$

With differentiation we obtain,

$$g'_n(z) = (\frac{1}{n})g_{n-1}(z) + (\frac{z+n-1}{n})g'_{n-1}(z), \tag{15}$$

and at $z = 1$,

$$g'_n(1) = \frac{1}{n} + g'_{n-1}(1), \tag{16}$$

since $g_{n-1}(1) = 1$. [Note that $g_{n-1}(1) = P_{(n-1)0} + P_{(n-1)1} + ...$ is the sum of all possible probabilities.]

Combining equation 12 with the above result we obtain,

$$E(|MAX(S)|) = g'_n(1) = \frac{1}{n} + g'_{n-1}(1). \tag{17}$$

Thus, by solving the above recurrence, with $g'_1(1) = \sum_i iP_{ni} = \sum_i iP_{1i} = 1$ (see equation 12), we have,

$$E(|MAX(S)|) = \frac{1}{n} + \frac{1}{n-1} + ... + \frac{1}{2} + 1 = H_n^{(1)}, \tag{18}$$

which is the n-th harmonic number $H_n^{(1)}$. [Note that, $H_n^{(1)} = 1 + \sum_{k=2}^{n} \frac{1}{n} \leq 1 + \int_{k=1}^{n} \frac{1}{n} = 1 + \ln n$.] We can proceed similarly to obtain the following result for the variance [104].

$$Var(|MAX(S)|) = H_n^{(1)} - H_n^{(2)}, \tag{19}$$

where $H_n^{(r)} = \frac{1}{n^r} + \frac{1}{(n-1)^r} + ... + \frac{1}{2^r} + \frac{1}{1^r} = \sum_{k=1}^{n} \frac{1}{k^r}$.

Obviously, a similar procedure can be carried out for vector sets in E^d for $d \geq 3$. Nevertheless, in most practical situations we are not interested in the exact values of $E(|MAX(S)|)$ and $Var(|MAX(S)|)$ but merely on dependable estimates (upper bounds) of their values. A simple and reliable upper bound on the value of $E(|MAX(S)|) = O((\ln n)^{d-1})$ is shown in [26], and in [58] this result is extended with the following theorem.

Theorem 5.4 *Let $S = (v_1, v_2, ..., v_n)$ be independently distributed E^d-valued random vectors, and let $H(S) = H(v_1, v_2, ..., v_n)$ be a subset of $(v_1, v_2, ..., v_n)$ with cardinality $|H(S)|$, invariant under permutations of the data - i.e. $H(v_1, ..., v_n) = H(v_{\sigma(1)}, ..., v_{\sigma(n)})$, for all $(v_1, ..., v_n) \in E^d$, and all permutations $(\sigma(1), ..., \sigma(n))$ - and possessing the inclusion property - i.e. $v_1 \in H(v_1, ..., v_n)$ implies $v_1 \in H(v_1, ..., v_i)$, for all $i \leq n$ - then,*

$$E(|H(S)|^2) \leq 5E(|H(S)|)^2,$$

for any fixed d.

Proof: See [58]. ■

Examples of geometric arrangements (beyond the set of all maximal vectors) that satisfy the conditions of theorem 5.4 - invariant under permutations of the data, and possession of inclusion property - include the convex hull, the set of isolated points, the lower envelope of straight line segments and other geometric structures.

It is, therefore, easy to verify that,

Corollary 5.3 *In a set S of n identically distributed random vectors in d-space, that satisfy the conditions of theorem 5.4, $O(Var(|MAX(S)|)) = O(E(|MAX(S)|)^2)$ for any fixed d.*

Proof: Substituting the result of theorem 5.4 in the definition of the variance [133] we have,

$$\begin{aligned} Var(|MAX(S)|) &= E(|MAX(S)|^2) - E(|MAX(S)|)^2 \\ &\leq 5E(|MAX(S)|)^2 - E(|MAX(S)|)^2 \\ &= 4E(|MAX(S)|)^2. \end{aligned} \tag{20}$$

Hence, we have $O(Var(|MAX(S)|)) = O(E(|MAX(S)|)^2)$. ■

5.3.2 The Convex Hull Algorithm

In this section we present a high-probability BSP algorithm for solving the E^d convex hull problem [36, 53] for any fixed d. The algorithm is very simple in concept, easy to implement, and has an optimal time complexity. The fundamental algorithmic idea derives from [56], however, the analysis of the algorithm in this section is substantially different from that in [56].[30] We show that our algorithm applies to any identically distributed continuous random vector distribution (the analysis of [56] is limited to the uniform distribution).[31] Furthermore, our algorithm achieves superior probability bounds.

The following lemma was introduced in section 5.2.2.

Lemma 5.5 *The convex hull $CH(S)$ resulting from the union[32] of $CH(S_1) \cup CH(S_2) \cup ... \cup CH(S_i)$ is equal to the convex hull of the union of $CH(S_1 \cup S_2 \cup ... \cup S_i)$, i.e. the convex hull of S is invariant under permutations of $(S_1, S_2, ..., S_i)$, and possesses the inclusion property.*

Proof: See section 5.2.2. ■

Assume that the set S of n points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly). Now we can proceed with the convex hull algorithm.

---

[30] The analysis of the *generic hull* algorithm in [56] concealed several inaccuracies.

[31] In the remainder of this section we shall assume that the input points are drawn independently from any identically distributed continuous distribution.

[32] An extended notion of the term *union* is used to accommodate convex hulls.

Algorithm 5.3

Input: *An array* $S_{i=1...n}$ *of* E^d *vectors.*
Output: *An ordered array* $CH_{i=1...n}$*, containing the convex hull* $CH(S)$ *of* S*.*

Stage–1 Each processor p_i computes sequentially the convex hull $CH(S_i)$ of the points in its possession.

Stage–2 The resulting convex hull points are concentrated into the minimum number of processors (possibly one), as follows: each processor counts the number of points on its convex hull, and an all-prefix operation is then initiated on the number of points per hull and per processor to find the processor destination of each point.

Stage–3 Each processor knows where to route every point in its possession, and performs this routing.

Stage–4 Stages 1, 2 and 3 are repeated until all convex hull points are concentrated in one processor.

Stage–5 The convex hull problem is computed sequentially on one processor.

The correctness of the algorithm follows directly from lemma 5.5. Note that, the algorithm may fail to terminate. As we shall show in the rest of this section, the probability of failure is so small that it doesn't seem worthwhile to invest much effort to treat it. For instance, a trivial solution would be that of employing a deterministic sequential algorithm (implemented by message passing [56]), whenever stages 2 to 4 fail more than t times to concentrate the points into one processor, where t is a constant.

The time complexity of algorithm 5.3 can be analyzed as follows. We start with the following lemma.

Lemma 5.6 *Let* $(S = v_1, v_2, ..., v_n)$ *be identically distributed* E^d*-valued random vectors, and let* $H(S) = H(v_1, v_2, ..., v_n)$ *be a subset of* $(v_1, v_2, ..., v_n)$ *with cardinality* $|H(S)|$*, invariant under permutations of the data - i.e.* $H(v_1, ..., v_n) = H(v_{\sigma(1)}, ..., v_{\sigma(n)})$*, for all* $(v_1, ..., v_n) \in E^d$*, and all permutations* $(\sigma(1), ..., \sigma(n))$ *- and possessing the inclusion property - i.e.* $v_1 \in H(v_1, ..., v_n)$ *implies* $v_1 \in H(v_1, ..., v_i)$*, for all* $i \leq n$ *- then,*

$$Prob[|H(S)| \geq \beta E(|H(S)|)] \leq 5e^{2\beta E(|H(S)|)(1-\frac{1}{\beta}-\ln\beta)},$$

for any $\beta > 1$*.*

Proof: We can view the cardinality $|H(S)|$ of the set $H(S)$ as being determined by n Bernoulli trials $(X_1, X_2, ..., X_n)$. Success $(X_i = 1)$ occurs in the i-th trial when $v_i \in H(v_1, v_2, ..., v_n)$. Obviously, $|H(S)| = X_1 + X_2 + ... + X_n$ and $E(|H(S)|) = P_1 + P_2 + ... + P_n$, where $P_i = Prob[X_i = 1]$, for all $1 \leq i \leq n$.

Since for any $\rho > 0$ the function $e^{\rho x}$ is strictly increasing in x,

$$Prob[|H(S)| \geq \beta E(|H(S)|)] = Prob[e^{\rho|H(S)|} \geq e^{\rho\beta E(|H(S)|)}], \tag{21}$$

where ρ will be determined later. Using Chebyshev's inequality and the result of theorem 5.4 we obtain,

$$Prob[|H(S)| \geq \beta E(|H(S)|)] \leq \frac{5E(e^{\rho|H(S)|})^2}{e^{2\rho\beta E(|H(S)|)}}. \tag{22}$$

The bulk of the proof consists of bounding $E(e^{\rho|H(S)|})^2$ and substituting a suitable value for ρ.

We first observe that, by the definition of expectation,

$$\begin{aligned} E(e^{\rho X_i}) &= e^{1\rho} Prob[X_i = 1] + e^{0\rho}(1 - Prob[X_i = 1]) \\ &= e^{\rho} Prob[X_i = 1] + 1 - Prob[X_i = 1] \\ &= 1 + Prob[X_i = 1](e^{\rho} - 1) \\ &= 1 + P_i(e^{\rho} - 1) \\ &\leq e^{P_i(e^{\rho}-1)}, \end{aligned} \tag{23}$$

since $1 + x \leq e^x$ for all x. Furthermore, since the X_i's are independent, we know that,

$$\begin{aligned} E(e^{\rho|H(S)|}) &= E(e^{\rho(X_1+X_2+\ldots+X_n)}) \\ &= E(e^{\rho X_1} e^{\rho X_2} \ldots e^{\rho X_n}) \\ &= E(e^{\rho X_1}) E(e^{\rho X_2}) \ldots E(e^{\rho X_n}) \\ &\leq e^{P_1(e^{\rho}-1)} e^{P_2(e^{\rho}-1)} \ldots e^{P_n(e^{\rho}-1)} \\ &= e^{(P_1+P_2+\ldots+P_n)(e^{\rho}-1)} \\ &= e^{E(|H(S)|)(e^{\rho}-1)}. \end{aligned} \tag{24}$$

Substituting for $E(e^{\rho|H(S)|})$ in inequality 22, we obtain,

$$Prob[|H(S)| \geq \beta E(|H(S)|)] \leq \frac{5e^{2E(|H(S)|)(e^{\rho}-1)}}{e^{2\rho\beta E(|H(S)|)}}. \tag{25}$$

This probability is minimized by setting $\rho = \ln \beta$, whereupon we find that,[33]

[33]Let $f(\rho) = \frac{5e^{2E(|H(S)|)(e^{\rho}-1)}}{e^{2\rho\beta E(|H(S)|)}}$.

$$\begin{aligned} f'(\rho) &= \frac{d}{d\rho}\left[\frac{5e^{2E(|H(S)|)(e^{\rho}-1)}}{e^{2\rho\beta E(|H(S)|)}}\right] \\ &= \frac{d}{d\rho}[2E(|H(S)|)(e^{\rho} - 1) - 2\rho\beta E(|H(S)|)] \frac{5e^{2E(|H(S)|)(e^{\rho}-1)}}{e^{2\rho\beta E(|H(S)|)}} \\ &= (2E(|H(S)|)e^{\rho} - 2\beta E(|H(S)|)) \frac{5e^{2E(|H(S)|)(e^{\rho}-1)}}{e^{2\rho\beta E(|H(S)|)}}. \end{aligned} \tag{26}$$

Hence, since $f'(\rho) = 0$ at $\rho = \ln \beta$ and $f''(\rho) > 0$ we have that $f(\rho)$ minimizes at $\rho = \ln \beta$.

$$
\begin{aligned}
Prob[|H(S)| \geq \beta E(|H(S)|)] &\leq \frac{5e^{2E(|H(S)|)(e^{\rho}-1)}}{e^{2\rho\beta E(|H(S)|)}} \\
&\leq \frac{5e^{2E(|H(S)|)(e^{\ln\beta}-1)}}{e^{2\ln\beta\beta E(|H(S)|)}} \\
&= 5e^{2\beta E(|H(S)|)(1-\frac{1}{\beta}-\ln\beta)}. \qquad (27)
\end{aligned}
$$

■

Examples of geometric arrangements that satisfy the conditions of lemma 5.6 (i.e. invariant under permutations of the data, and possession of inclusion property) include the convex hull, the collection of all maxima vectors, the set of isolated points, the lower envelope of straight line segments and other geometric structures.

When $H(S)$ is the convex hull of $S = (v_1, v_2, ..., v_n)$, the distribution of $H(S)$ is generally hard to find. For many distributions, however, the asymptotic behaviour of $E(|H(S)|)$ is known. Among the known results, we cite [23, 140, 145, 146, 147]:

- Normal Distribution in E^2, $E(CH) = 2\sqrt{2\pi \log n}$
- Normal Distribution in E^d, $E(CH) = O((\log n)^{\frac{d-1}{2}})$
- Uniform Distribution in $[1,0]^d$, $E(CH) = O((\ln n)^{d-1})$
- Uniform Distribution in k-gon, $E(CH) = \frac{2k}{3}\log n$

The following, more general result, is due to [26].

Lemma 5.7 *The expected number of convex hull points in a point set of n points in E^d having their components chosen independently from any set of continuous distribution functions (possibly different) is bounded above by $O((\ln n)^{d-1})$ for any fixed d.*

Proof: Every convex hull point is a maximal point under at least one of the 2^d possible distinct assignments of $+$ and $-$ to the d-dimensions. Since the average number of maximal points in each of the 2^d assignments is bounded above by $O((\ln n)^{d-1})$ [26], the total number of expected convex hull points is bounded above by

$$
2^d O((\ln n)^{d-1}) = 2O((2\ln n)^{d-1}) = O((\ln n)^{d-1}). \qquad (28)
$$

We thus have the above lemma. ■

Furthermore, from lemma 5.6 it follows that,

Lemma 5.8 *With probability at least $1 - O(\frac{1}{n^{\ln\ln n}})$, we have that,*

$$
n_{(j)} \leq p_{(j-1)} \ln n E(CH(S)),
$$

where $n_{(j)}$ is the number of remaining points after the j-th iteration *(the term* iteration *refers to stage 4 of algorithm 5.3) and $p_{(j-1)}$ is the number of active processors after iteration $(j-1)$.*

Proof: Let P be the probability that $\max_i \{|CH(S_i)|\} \geq \ln n E(|CH(S)|)$, where $0 \leq i < p_{(j-1)}$, and S_i is the set of points on processor i. From lemma 5.6 we have that,

$$\begin{aligned} P &= Prob[\max_i \{|CH(S_i)|\} \geq \ln n E(|CH(S)|)] \\ &\leq p5e^{2 \ln n E(|H(S)|)(1-\frac{1}{\ln n} - \ln \ln n)} \\ &\leq n5e^{2 \ln n E(|H(S)|)(1-\frac{1}{\ln n} - \ln \ln n)} \\ &\leq \frac{5n}{n^{2(\ln \ln n + \frac{1}{\ln n} - 1)}} \\ &= O(\frac{1}{n^{\ln \ln n}}). \end{aligned} \tag{29}$$

Therefore, with probability at least $1 - O(\frac{1}{n^{\ln \ln n}})$, we have that,

$$\begin{aligned} n_{(j)} &\leq p_{(j-1)} \max_i \{|CH(S_i)|\} \\ &\leq p_{(j-1)} \ln n E(|CH(S)|). \end{aligned} \tag{30}$$

■

Thus, it follows with probability at least $1 - O(\frac{1}{n^{\ln \ln n}})$ that after the first iteration we obtain a point set of size,

$$n_{(1)} \leq p \ln n E(|CH(S)|). \tag{31}$$

Concentrating the convex hull points into the minimum number of processors, results in $p_{(1)}$ active processors, where,

$$p_{(1)} = \lceil \frac{n_{(1)}}{\frac{n}{p}} \rceil. \tag{32}$$

It is straightforward to verify that,

$$p_{(j)} = \lceil \frac{n_{(j)}}{\frac{n}{p}} \rceil. \tag{33}$$

Thus, by combining the above equation with the result of lemma 5.8 we obtain the following recurrence,

$$n_{(j)} \leq \frac{n_{(j-1)} p \ln n E(|CH(S)|)}{n} \tag{34}$$

This formula will determine $n_{(j)}$ if we provide the initial condition $n_{(0)} = n$. Solving the recurrence we obtain,[34]

$$n_{(j)} \leq \frac{(p \ln n E(|CH(S)|))^j}{n^{j-1}}, \tag{35}$$

[34]In order to simplify exposition, we assume that $\lceil \frac{n_{(j)}p}{n} \rceil = \frac{n_{(j)}p}{n}$.

where j is the number of iterations. Undoubtedly, if algorithm 5.3 is to terminate within k iterations, $n_{(k)}$ should be less than $\frac{n}{p}$. Solving with respect to k,

$$\begin{aligned}
\frac{n}{p} &\geq \frac{(p \ln n E(|CH(S)|))^k}{n^{k-1}} \\
p &\leq (\frac{n}{p \ln n E(|CH(S)|)})^k \\
\ln p &\leq k \ln \frac{n}{p \ln n E(|CH(S)|)} \\
k &\geq \frac{\ln p}{\ln \frac{n}{p \ln n E(|CH(S)|)}} \\
&\geq \frac{\ln p}{\ln n - \ln p - \ln \ln n - \ln E(|CH(S)|)}. \qquad (36)
\end{aligned}$$

Lemma 5.9 *With probability* $1-O(\frac{1}{n^{\ln \ln n}})$, $n_{(k)} \leq \frac{n}{p}$ *for any* $k \geq \ln p/(\ln n - \ln p - \ln \ln n - O((d-1)\ln \ln n))$, *where* $n_{(k)}$ *is the problem size after the* k*-th iteration (provided that* $\ln n - \ln p - \ln \ln n - O((d-1)\ln \ln n) > 0$*).*

Proof: Let in inequality 36 $E(|CH(S)|) = O((\ln n)^{d-1})$ (see lemma 5.7). ■

Note that, $n_{(k)} \leq \frac{n}{p}$ indicates that algorithm 5.3 will terminate within k iterations. It is now straightforward to verify, for the three-dimensional case, the following lemma.

Lemma 5.10 *With probability at least* $1 - O(\frac{1}{n^{\ln \ln n}})$, $n_{(k)} \leq \frac{n}{p}$ *for any* $k \geq \frac{\ln p}{\ln n - \ln p - \ln \ln n - 2 \ln 2 \ln n}$, *where* $n_{(k)}$ *is the three-dimensional problem size after the* k*-th iteration (provided that* $\ln n - \ln p - \ln \ln n - 2 \ln 2 \ln n > 0$*).*

Proof: In [23] it is proved that (for three-dimensions),

$$E(|CH(S)|) \leq 2^3(\frac{(H_n^{(1)})^2 + H_n^{(2)}}{2}), \qquad (37)$$

where $H_n^{(r)} = \sum_{k=1}^{n} \frac{1}{k^r}$.

In section 5.3.1 it was shown that $H_n^{(1)} \leq 1 + \ln n$. [Note that, $H_n^{(1)} = \ln n + \gamma + O(\frac{1}{\ln^2 n})$, γ being Euler's constant [149].] Moreover, it is not difficult to see that $H_n^{(2)}$ is bounded above by a constant, i.e. $H_n^{(2)} = \sum_{k=1}^{n} \frac{1}{k^2} = 1 + \sum_{k=2}^{n} \frac{1}{k^2} \leq 1 + \int_1^{\infty} \frac{1}{k^2} dk = 2$.

With substitution we obtain,

$$E(|CH(S)|) <\approx 4 \ln^2 n. \qquad (38)$$

Combining the above result with inequality 36 we obtain,

$$k \geq \frac{\ln p}{\ln n - \ln p - \ln \ln n - 2 \ln 2 \ln n} \qquad (39)$$

We thus have the above lemma. ■

This implies immediately,[35]

Theorem 5.5 *There exists a BSP algorithm that, given a set S of n points in E^3, can construct the convex hull $CH(S)$ of S in*

$$O(k(\frac{n}{p}\log\frac{n}{p} + \log_t p \max\{tg, L\} + \max\{g\frac{n}{p}, L\}))$$

time, with probability $1 - O(\frac{1}{n^{\ln\ln n}})$, for any $k \geq \frac{\ln p}{\ln n - \ln p - \ln\ln n - 2\ln 2\ln n}$ (provided that $\ln n - \ln p - \ln\ln n - 2\ln 2\ln n > 0$).

Proof: The sequential solution of the convex hull problem in stages 1 and 5 takes time $O(\frac{n}{p}\log\frac{n}{p})$. The all-prefix operation in stage 2 takes time $O(\frac{n}{p} + \log_t p \max\{tg, L\})$ (see section A.2.3). Finally, the time required for the routing operation (stage 3) is $O(\max\{g\frac{n}{p}, L\})$. We thus obtain the above result. ■

Accordingly,

Corollary 5.4 *The three-dimensional convex hull problem can be solved in optimal*

$$O(\frac{n\log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(n^\epsilon)$, $L = O(\frac{n\log n}{p\log p})$, and $g = O(\log n)$, with probability at least $1 - O(\frac{1}{n^{\ln\ln n}})$, for any constant $0 < \epsilon < 1$.

Proof: Let in theorem 5.5 $p = n^\epsilon$, for any constant $0 < \epsilon < 1$.

$$\begin{aligned} k &\geq \frac{\ln n^\epsilon}{\ln n - \ln n^\epsilon - \ln\ln n - 2\ln 2\ln n} \\ &= \frac{\epsilon\ln n}{\ln n - \epsilon\ln n - \ln\ln n - 2\ln 2\ln n} \\ &= \Omega(\frac{\epsilon\ln n}{(1-\epsilon)\ln n}) \\ &= \Omega(\frac{\epsilon}{1-\epsilon}) \\ &= \Omega(1), \end{aligned} \tag{40}$$

which implies that only a constant number of iterations ($\approx \frac{\epsilon}{1-\epsilon}$) are necessary for the algorithm to terminate, with probability at least $1 - O(\frac{1}{n^{\ln\ln n}})$. Moreover, letting in theorem 5.5 $t = 2$, we obtain,

[35] The assumption is that an optimal $O(n\log n)$ sequential three-dimensional convex hull algorithm exists, i.e. [137].

Problem Size	*Theoretical Upper bound*	*Observed*
2^{12}	4	2
2^{16}	2	1
2^{22}	1	1

Table 2: Number of iterations for the $2D$ convex hull problem of varying size with a fixed number of nodes (8) of an IBM SP1 multicomputer.

$$\begin{aligned}
T &= O(k(\frac{n}{p}\log\frac{n}{p} + \log p \max\{g, L\} + \max\{g\frac{n}{p}, L\})) \\
&= O(\frac{n}{p}\log\frac{n}{p} + \log p \max\{g, L\} + \max\{g\frac{n}{p}, L\}) \\
&= O(\frac{n\log n}{p}), \qquad (41)
\end{aligned}$$

provided that $L = O(\frac{n\log n}{p\log p})$, and $g = O(\log n)$.

We thus have the above corollary. ■

[Note that, for all large enough $n \gg p$, k tends to one; in practical situations k is in the range [1...3] (see table 2).]

Similar results may be obtained for convex hull problems in higher dimensions. As noted in [56], it is an interesting open problem to give a probabilistic analysis for data distributions that do not conform to the assumption of identically distributed random vectors.

[To demonstrate the practical relevance of our scalable BSP algorithms, we implemented the above convex hull algorithm on a Silicon Graphics Power Challenge shared memory system (consisting of four 64-bit Streaming Superscalar TFP processors) and an IBM 9076 close-coupled distributed memory system (consisting of eight IBM RS/6000 model 580 processors). The results of our experiments can be found in appendix C section C.4.]

5.4 A Simple Technique for Improving the Performance of Convex Hull Algorithms

This section is concerned with a simple, rather natural technique (preprocessing step), that is likely to enhance the performance of any convex hull algorithm. The pre-processing step resembles to one step of Quickhull [63], that is, it identifies and eliminates roughly all the points internal to the hull, under suitable assumptions. The algorithm is remarkably simple and highly efficient, and its output can be fed into any convex hull algorithm.

In [79] it is shown that a simple sequential algorithm (a simplified and practical version of [59]) could eliminate all but $O(n^{\frac{d-1}{d}})$ uniformly distributed vectors in the unit d-cube, where d is the index of the dimension (note that, superior results hold for multivariate normal and t-distributions). In the remainder of this section we show an efficient BSP adaption of the sequential algorithm.

We start by recalling the idea of the two-dimensional algorithm. Suppose that a set S contains n independently distributed random points each of which is uniformly distributed in the unit square. Pick any four points from S. These points are the vertices of a quadrilateral Q (in the degenerate case Q is a triangle). By its definition Q is contained within the convex hull of S, thus, we can eliminate all points that are inside Q. The goal of the algorithm is to find the four points such that after the elimination step only a few will on average remain. Clearly, the process should be computationally simple and inexpensive, otherwise it is not warranted. In [79], it is shown that a simple and highly efficient choice is the selection of the four points (z_1, z_2, z_3, z_4) that minimize the four functions $(x+y, -x+y, x-y, -x-y)$ (to secure uniqueness choose the points with the lowest indices that minimize these functions) . The last step is to eliminate those points that are internal to the smallest rectangle R defined by the horizontal and vertical lines passing through (z_1, z_2, z_3, z_4).

If the n points are distributed in the unit d-cube, the above procedure is expected to eliminate all but $O(n^{\frac{d-1}{d}})$ points. In particular, for the two-dimensional case, the above procedure is expected to eliminate all but $8\sqrt{n}$ points [79].

Assume that the set S of n points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 5.4

Input: *An array* $S_{i=1\ldots n}$ *of* E^2 *vectors.*
Output: *An array* $H_{i=1\ldots k}$, *containing "candidate" convex hull vertices, where* $k \leq n$.

Stage–1 Each processor picks, from the points in its possession, the four points that minimize the functions $x+y, -x+y, x-y, -x-y$.

Stage–2 A reduce operation is initiated, on the points of the previous stage to determine the four points that (globally) minimize the functions $x+y, -x+y, x-y, -x-y$.

Stage–3 Each processor eliminates the points in its possession that fall into the rectangle defined by the four points selected in stage 2.

The time complexity of the algorithm can be analyzed as follows. The local computation in stages 1 and 3 can be performed in time $O(\frac{n}{p})$. The reduce operation in stage 2 takes $O(\log_t p \max\{tg, L\})$ time steps. Thus, it is straightforward to verify that the time complexity of algorithm 5.4 is

$$O(\frac{n}{p} + \log_t p \max\{tg, L\}). \tag{42}$$

The above result affords us with the following powerful theorem.

Theorem 5.6 *The combination of algorithm 5.4 with any two-dimensional* $O(\frac{n^2}{p})$ *worst-case time convex hull algorithm has an expected*

$$O(\frac{n}{p})$$

running time, on the BSP model, for any integers n, p, L, and g such that $L = O(\frac{n}{p \log_t p})$, *and* $g = O(\frac{n}{tp \log_t p})$, *under the assumption that the input vectors are uniformly distributed over* $[0,1]^2$ *(unit square).*

Proof: Algorithm 5.4 terminates in time $O(\frac{n}{p})$ and eliminates (on the average) all but $O(\sqrt{n})$ points. Since, by definition, the convex hull worst-case running time is $O(\frac{n^2}{p})$, the overall expected-time complexity will be $O(\frac{n}{p}) + O(\frac{(\sqrt{n})^2}{p})$ which is $O(\frac{n}{p})$. ■

In the same vein, the combination of algorithm 5.4 with any $O(\frac{n \log n}{p})$ worst-case time two-dimensional convex hull algorithm has an expected $O(\frac{n}{p})$ running time. Therefore,

Corollary 5.5 *The combination of algorithm 5.4, with the algorithms of sections 5.2.1 and 5.2.2 has an expected*

$$O(\frac{n}{p})$$

running time, on the BSP model, for any integers n, p, L, and g such that $L = O(\frac{n}{p \log_t p})$, *and* $g = O(\frac{n}{tp \log_t p})$, *under the assumption that the input vectors are uniformly distributed over* $[0,1]^2$ *(unit square).*

Proof: Immediate. ■

Similar results may be obtained for convex hull problems in higher dimensions.
[To demonstrate the practical relevance of our scalable BSP algorithms, we implemented the above algorithmic technique on a Silicon Graphics Power Challenge shared memory system (consisting of four 64-bit Streaming Superscalar TFP processors) and an IBM 9076 close-coupled distributed memory system (consisting of eight IBM RS/6000 model 580 processors). The results of our experiments can be found in appendix C section C.5.]

6 Maxima Problems

This section is concerned with a simple, rather natural extension of the convex hull, the problem of determining the maxima of a vector set [17, 11, 50]. This problem occurs in a considerable number of applications, in statistics,[36] economics and operations research. The problem can be precisely defined as follows. Given a set S of n vectors (points) in E^d, with coordinates $(x_1, x_2, ..., x_d)$, find those vectors in S that are not dominated by any other vector in S. The *dominance* is a relation on a set S of n vectors as follows.

[36] A survey of geometric techniques in statistics is given in [154]

Definition 6.1 *Vector u dominates vector v if $x_i(u) \leq x_i(v)$ for all $i = 1, 2, ..., d$.*

The relationship between the maxima and the convex hull problem was illustrated in section 5.3 (see lemma 5.7). Here, we shall just recall the idea. Every convex hull point is a maximal under at least one of the 2^d possible distinct assignments of $+$ and $-$ signs to the d-dimensions. Therefore, both the maxima and the convex hull are representations of the boundary of S, eventhough the maxima provides a cruder representation [137].

As we already did for the convex hull problem, we shall first state a lower bound for the two dimensional case.[37] In [107] it is shown that any sequential algorithm that solves the maxima problem in two-dimensions requires time $\Omega(n \log n)$. Undoubtedly, since any maxima problem in E^{d-1} can be considered as a problem in E^d, such a lower bound holds for all $d \geq 3$.

6.1 Maxima in the Plane

A simple, and rather natural solution to the two-dimensional maxima problem derives directly from its definition - solve the dominance counting problem and count the number of vectors that are not dominated by any other vector. Hence, we obtain the following algorithm.

Algorithm 6.1

Input: *An array $S_{i=1...n}$ of E^2 vectors.*
Output: *The number of maximal vectors $MAX(S)$ in S.*

> **Stage–1** Solve the dominance counting problem with respect to S (see section 3.2.1). [Note that, the algorithm of section 3.2.1 should be modified to report the number of vectors that dominate a certain vector v, instead of reporting the number of vectors dominated by v.]
>
> **Stage–2** Each processor scans over the vectors in its possession and determines the number of vectors that are not dominated by any other vector.
>
> **Stage–3** A reduce operation is initiated (on the totals of stage 2), using the binary operator $+$ (*plus*).

The correctness of the algorithm follows directly from the definition of the maxima problem. Algorithm 6.1 affords us with the following result.

Theorem 6.1 *There exists a BSP algorithm that, given a set S of n vectors in the plane, can determine the number of maximal vectors in*

[37] It is clear that, the one-dimensional maxima problem reduces to finding the maximal element of a linear set (see section A.2.2 for a BSP algorithm). Obviously, if duplicate elements do not exist, the number of maximal elements in one-dimension is one.

$$
\begin{aligned}
O(\quad & p\ln^2 n \log(p\ln^2 n) + \\
& \frac{n}{p}\log p + \\
& \max\{gp\ln^2 n, L\} + \\
& \log_t p \max\{tgp, L\} + \\
& \max\{g\kappa\frac{n}{p}, L\} + \\
& \kappa\frac{n}{p}\log(\kappa\frac{n}{p}))
\end{aligned}
$$

time - with probability at least $1 - O(\frac{1}{n})$ *- where* $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The local computation in stage 2 and the reduce operation in stage 3 take time $O(\frac{n}{p} + \log_t p \max\{tg, L\})$ and are undoubtedly dominated by the dominance counting time complexity of stage 1 (see theorem 3.2). We thus obtain the above result. ■

The following note deals with the notion of optimality.

Corollary 6.1 *The two-dimensional maxima problem can be solved in optimal*

$$O(\frac{n\log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n\log n}{p\log p})$, *and* $g = O(\log n)$.

Proof: Let in theorem 6.1 $t = 2$. ■

Note that, the probabilistic feature of the algorithm can be eliminated if the dominance counting algorithm of section 3.2.1 is replaced by a similar deterministic algorithm (i.e., see algorithm B.4).

6.2 Maxima in Three Dimensions

In this section we present a BSP algorithm for solving the three-dimensional maxima problem - given a set S of n vectors in Euclidean space, determine the set $MAX(S)$ of maximal vectors, such that, given a vector u in $MAX(S)$ there is no vector v in S with $x(v) > x(u)$ and $y(v) > y(u)$ and $z(v) > z(u)$. The algorithm to be presented derives from the *3D-Maxima* algorithm of [55] and the following lemma.

Lemma 6.1 *The maxima set* $MAX(S)$ *resulting from the union*[38] *of* $MAX(S_1) \cup MAX(S_2) \cup ... \cup MAX(S_i)$ *is equal to the maxima set of the union of* $MAX(S_1 \cup S_2 \cup ... \cup S_i)$*, i.e. the maxima set of* S *is invariant under permutations of* $S_1, S_2, ..., S_i$*, and possesses the inclusion property.*

[38] An extended notion of the term *union* is used to accommodate maximal sets.

Proof: The correctness of the above lemma follows directly form the definition of the maxima problem. ■

Moreover, the following is shown in [55].

Lemma 6.2 *Let S be a set of n vectors, and let S_V be the set of vectors in a vertical slab delimited by two planes V and V', with $x(V) < x(V')$ and which are parallel to the yz-plane. Furthermore, let S_H be the set of vectors in a horizontal slab delimited by two planes H and H', with $z(H) < z(H')$ and which are parallel to the xy-plane. Then a vector u in $V - H$ is dominated by a vector v in $H - V$ if and only if it is also dominated by the intersection vector w of V' with the x-monotonic chain resulting from the two-dimensional maxima of the projection of S_H on the plane H.*

Proof: A detailed account of the above result is given in [55]. ■

The idea of the algorithm may be summarized as follows. Partition the Euclidean space in p vertical and horizontal slabs of approximately equal size $\frac{n}{p}$. Then, according to the above two lemmas a point v in S_{V*H} is either dominated by a point in S_V or in S_H or by one of the p intersection points in S_I which are induced by the other horizontal slabs. Assume that the input is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 6.2

Input: *An array $S_{i=1\ldots n}$ of E^3 vectors.*
Output: *The number of maximal vectors $MAX(S)$ in S.*

Stage–1 The bucketing algorithm of section B.2 is used to route the vectors in S to the appropriate bucket (processor), using as key the z-coordinate, which yields set S_{H_i} on processor p_i. Furthermore, let H_i be the plane parallel to the xy-plane passing through the vector in S_{H_i} with the minimum z-coordinate. [Notice that the resulting buckets are of size $< (1 + \frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–2 Each processor p_i solves the three-dimensional maxima problem for the vectors in S_{H_i}. Moreover, all non-maximal vectors are removed from S_{H_i}.

Stage–3 Each processor p_i projects the points in S_{H_i} to the plane H_i.

Stage–4 Each processor p_i computes the two-dimensional maxima of the projection of S_{H_i} to the plane H_i. Additionally, each processor p_i determines the x-monotonic chain C_i resulting from the two-dimensional maxima computed in the previous step.

Stage–5 The bucketing algorithm of section B.2 is used to route the vectors in S to the appropriate bucket (processor), using as key the

x-coordinate, which yields set S_{V_j} on processor p_j. [Notice that the resulting buckets are of size $< (1 + \frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–6 A global broadcast operation is initiated, where processor p_j sends vector v_j, the vector in S_{V_j} with maximum x-coordinate, to all other processors. Furthermore, let V_j be the plane parallel to the yz-plane passing through v_j.

Stage–7 Each processor p_i determines the intersection vector I_{ij} of C_i with the planes V_j, for $0 \leq j < p$. Note that, there can be at most p intersections.

Stage–8 A global broadcast operation is initiated, where processor p_i sends the set of intersection vectors I_{ij} to processor p_j, for $0 \leq j < p$. Obviously, no processor will send or receive more than p vectors.

Stage–9 Each processor p_j solves the three-dimensional maxima problem for the vectors in $S_{V_i} + I_{ij}$, for $0 \leq i < p$.

The correctness of the algorithm follows directly from lemmas 6.1 and 6.2.

Theorem 6.2 *There exists a BSP algorithm that, given a set S of n points in the E^3, can compute the three-dimensional maxima set $MAX(S)$ of S in*

$$
\begin{aligned}
O(\quad & p \ln^2 n \log (p \ln^2 n) + \\
& \frac{n}{p} \log p + \\
& \max \{gp \ln^2 n, L\} + \\
& \log_t p \max \{tgp, L\} + \\
& \max \{g\kappa \frac{n}{p}, L\} + \\
& (p + \kappa \frac{n}{p}) \log (p + \kappa \frac{n}{p}))
\end{aligned}
$$

time - with probability at least $1 - O(\frac{1}{n})$ - where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The bucketing algorithm in stages 1 and 5 requires $O(p \ln^2 n \log (p \ln^2 n) + \frac{n}{p} \log p + \max \{gp \ln^2 n, L\} + \log_t p \max \{tgp, L\} + \max \{g\kappa \frac{n}{p}, L\})$ time (see theorem B.3). The sequential solution of the two and three-dimensional maxima problem in stages 2 and 4 can be accomplished in $O(\kappa \frac{n}{p} \log (\kappa \frac{n}{p}))$ time steps. The projection of the vectors in stage 3 and the determination of the x-monotonic chain in stage 4 take time $O(\kappa \frac{n}{p})$. The communication complexity in stages 6 and 8 is bounded above by $O(\max \{gp, L\})$. The determination of the intersection set in stage 7 requires $O(\frac{n}{p})$ time steps. Finally, the sequential solution of the three-dimensional maxima problem in stage 9 takes time $O((p + \kappa \frac{n}{p}) \log (p + \kappa \frac{n}{p}))$. ■

Accordingly,

Corollary 6.2 *The three-dimensional maxima problem can be solved in optimal*

$$O(\frac{n \log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n \log n}{p \log p})$, and $g = O(\log n)$.

Proof: Let in theorem 6.2 $t = 2$. ■

7 Visibility

Given a set S of "forbidden" curves and a point p in E^2, a point q is *visible* from p if the line segment $\overline{pq}$ does not intersect any forbidden curve in S [15, 82, 51, 52]. Normally, visibility is defined with respect to a point p. In the plane, if S consists of line segments, the locus of points visible from p form a (possibly unbounded) star-shaped polygon, called the *visibility polygon* [19, 21]. Visibility problems occur in a multiplicity of applications in graphics, scene analysis and robotics. In the following section an algorithm for the two-dimensional lower envelope problem is developed that can be employed to solve some of the most fundamental visibility problems. In particular, in section 7.1 it is shown that, it suffices to compute the lower envelope of a set of segments to obtain the visibility polygon from any finite point.

7.1 Lower Envelope

Let S be a set of n non-intersecting line segments $(s_1, s_2, ..., s_n)$, except possibly at their endpoints, and p be a point at negative infinity below all the line segments. The visibility polygon can now be defined by the *lower envelope* of the n segments. More formally, the lower envelope of a set S of n segments, is defined to be the set S' consisting of the visible portions of the segments from negative infinity (see figure 5). Note that, an algorithm for the lower envelope can be employed to handle the case when p is any finite point, except that the view of segments being ordered by x-coordinate is replaced by the view that segments are ordered radially around p [143]. Therefore, it suffices to compute the lower envelope of a set of segments to establish a method for determining the visibility from a point. In the remainder of this section we show how the lower envelope of a set S of line segments can be constructed on the BSP model in optimal time. The basic idea behind the algorithm of this section is illustrated in the following two lemmas.

Lemma 7.1 *The lower envelope $LE(S)$ resulting from the union*[39] *of $LE(S_1) \cup LE(S_2) \cup ... \cup LE(S_i)$ is equal to the lower envelope of the union of $LE(S_1 \cup S_2 \cup ... \cup S_i)$, i.e. the lower envelope of S is invariant under permutations of $(S_1, S_2, ..., S_i)$, and possesses the inclusion property.*

[39] An extended notion of the term *union* is used to accommodate lower envelopes.

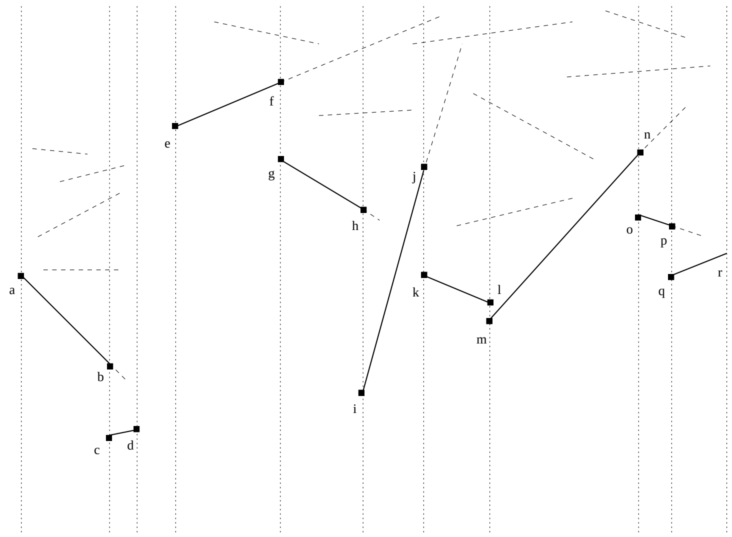

Figure 5: Two-dimensional lower envelope; visible portions of non-intersecting straight line segments from $-\infty$.

Proof: The correctness of the above lemma follows directly from the definition of the lower envelope (see preceding text). ■

Furthermore,

Lemma 7.2 *The lower envelope of n line segments is x-monotonic,*[40] *and, if the line segments do not intersect it has a size of $O(n)$.*

Proof: The x-monotonicity of the lower envelope of a segment set follows directly from the definition of the lower envelope problem. Moreover, it is straightforward to verify that (see figure 6),

$$\begin{aligned} |LE(S_{(1)})| &= 1 \\ |LE(S_{(n)})| &<= |LE(S_{(n-1)})| + 2 = 2n - 1 = O(n), \end{aligned} \tag{43}$$

where $S_{(i)}$ denotes a segment set of size i, i.e. $|S_{(i)}| = i$. Thus, we have the above lemma. ■

The algorithm that follows derives from a modified version of the lower envelope algorithm of [55] and the above lemmas. Since the algorithm to be presented employs a sequential lower envelope algorithm, it is assumed that an optimal $O(n \log n)$ sequential lower envelope algorithm exists [143]. Additionally, assume that the set

[40] A set S of segments is said to be x-monotonic if a line orthogonal to the x-axis intersects at most one segment in S (except at endpoints).

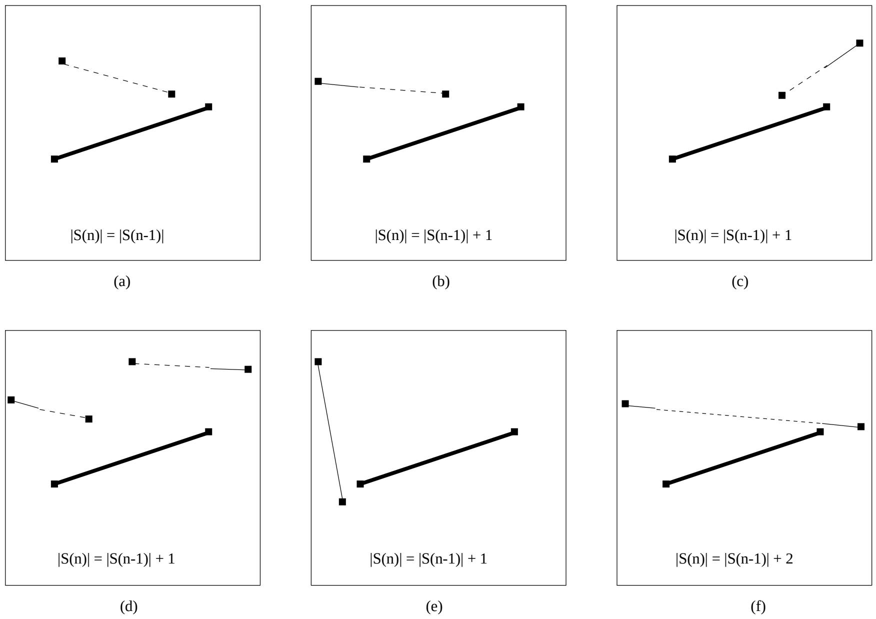

Figure 6: Addition of a new straight line segment.

S of n segments is initially distributed among the p processing units equally but possibly arbitrarily (uniformly). Now we can proceed with the lower envelope algorithm.

Algorithm 7.1

Input: *An array* $S_{i=1\ldots n}$ *of* E^2 *straight line segments.*
Output: *An ordered array* $LE_{i=1\ldots 2n}$ *containing the lower envelope* $LE(S)$ *of* S.

Stage–1 Each processor solves the lower envelope problem for the set S_i of segments in its possession, which results in the x-monotonic chain $LE^*(S_i)$ on processor p_i. [Note that, $|LE^*(S_i)| \leq 2\frac{n}{p} - 1 = O(\frac{n}{p})$.]

Stage–2 The bucketing algorithm of section B.2 is used to route the segments in $\cup_i LE^*(S_i)$ to the appropriate bucket (processor), using as key the x-coordinate of the right endpoint, which yields set S_i^* on processor p_i. [Notice that the resulting buckets are of size $< (1+\frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–3 A global broadcast operation is initiated, where processor p_i sends line l_i, the vertical line passing through the endpoint with maximum x-coordinate in S_i^*, to all other processors.

Stage–4 A global broadcast operation is initiated, where processor p_i sends segment s in $LE^*(S_i)$, to processor p_j, if and only if s intersects

l_j. [Note that, each processor p_i will receive a set R_i of (at most) p segments.]

Stage–5 Each processor p_i computes the lower envelope $LE(S_i^* \cup R_i)$ for the union of segments in S_i^* and R_i.

The correctness of the algorithm is easy to prove. Since, after the completion of stage 2 the segments in $S^* = \cup_i S_i^*$ are bucketed by the x-coordinate of their right endpoints, the lower envelope $LE(S_i^*)$ of some processor p_i can be affected by those segments R_i in processors $p_{i+1}...p_{p-1}$ that have their left endpoints to the left of the rightmost endpoint in S_i^*. Obviously, since after the completion of stage 1 the resulting chains are x-monotonic (see lemma 7.2) there can be at most p such segments (one per x-monotonic chain). Stage 4 identifies those segments, and, therefore, in stage 5 each processor correctly computes the lower envelope of its share of the global problem size.

Theorem 7.1 *There exists a BSP algorithm that, given a set S of n segments in the plane, can compute the lower envelope problem in*

$$O(\quad p\ln^2 n \log(p\ln^2 n) + \frac{n}{p}\log p + \max\{gp\ln^2 n, L\} + \log_t p \max\{tgp, L\} + \max\{g\kappa\frac{n}{p}, L\} + (p+\kappa\frac{n}{p})\log(p+\kappa\frac{n}{p}))$$

time - with probability at least $1 - O(\frac{1}{n})$ - where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The sequential solution of the lower envelope problem is stage 1 can be computed in $O(\frac{n}{p}\log\frac{n}{p})$ time steps. The bucketing algorithm in stage 2 requires $O(p\ln^2 n\log(p\ln^2 n) + \frac{n}{p}\log p + \max\{gp\ln^2 n, L\} + \log_t p\max\{tgp, L\} + \max\{g\kappa\frac{n}{p}, L\})$ time (see theorem B.3). The communication complexity in stages 3 and 4 is bounded above by $O(\max\{gp, L\})$. Finally, the sequential solution of the lower envelope problem in stage 2 can be computed in time $O((p+\kappa\frac{n}{p})\log(p+\kappa\frac{n}{p}))$. ■

Therefore,

Corollary 7.1 *The lower envelope problem can be solved in optimal*

$$O(\frac{n\log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n\log n}{p\log p})$, and $g = O(\log n)$.

Proof: Let in theorem 7.1 $t = 2$. ■

The same algorithmic idea can be employed to solve the problem for possibly intersecting line segments. Simply replace the sequential algorithm employed in stages 1 and 5 of the above algorithm, by a sequential algorithm that can handle intersecting line segments [89]. However, since the size of the resulting problem is no longer $O(n)$, some extra storage requirements are necessary. [Note that, the size of the lower envelope for possibly intersecting line segments is $O(n\alpha(n))$, where $\alpha()$ is the slow growing inverse Ackermann function [55].]

7.2 Three-Dimensional Hidden-Surface Removal

The generation of three-dimensional graphic scenes on a two-dimensional display screen requires, among other tasks, the removal of hidden-surfaces of objects (known as *hidden-surface elimination* problem) to obtain a realistic two-dimensional view of the three-dimensional scene. In this section we present an efficient algorithm for solving the three-dimensional hidden-surface problem based on a scan line approach.[41] In particular, each scan-line plane intersects a set of objects in three-dimensional space and yields a set of non-overlapping convex polygons [110]. Therefore, the problem of three-dimensional hidden-surface removal reduces to that of computing the visible edges of a set R of n non-overlapping convex polygons from a point q. In the remainder of this section we employ the techniques (algorithms) developed in the previous sections to solve the problem of determining the visible edges of a set of non-overlapping convex polygons.

The algorithm, which consists of two phases, derives from the visible edges sequential algorithm of [110] and the lower envelope algorithm of section 7.1. The first part deals with each polygon individually by locating its two supporting vertices (for details see [137]), hence, obtaining the potentially visible portion of the polygon. The second part determines, for each convex polygon, the portion of its potentially visible fragment that is not obscured by any other convex polygon (see figure 7).

Assume that the set R of n non-overlapping convex polygons is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 7.2

Input: *An array $R_{i=1\ldots n}$ of E^2 convex hulls and a reference point q.*
Output: *An array $R^*_{i=1\ldots n}$ containing the visible portion of S.*

[41] Scan devices usually scan the three-dimensional graphic scene from left to right and/or top to bottom in a pre-defined fixed pattern. The scan line algorithms are designed to generate the computed image line by line. As noted in [110], each scan line is considered as a lateral cut through the three-dimensional graphic scene where the scan line plane cuts faces of the scene into segments. The hidden surface elimination problem then reduces to a two-dimensional hidden line problem which requires the elimination of invisible segments by each scan line. Note that the result of the scan line method is (in most circumstances) a collection of non-overlapping simple polygons. We shall further assume that the polygons are convex.

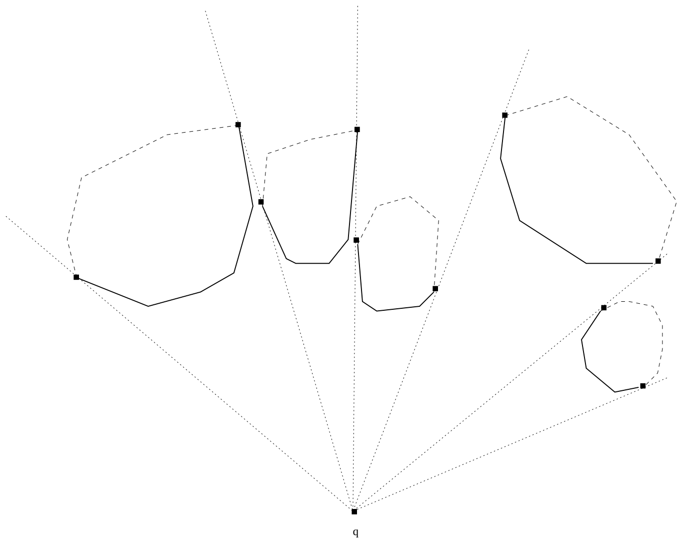

Figure 7: Visible segments of non-intersecting convex polygons from an arbitrary point.

Stage–1 Each processor computes the two supporting lines (with respect to q) of each polygon in its possession, thereby obtaining set R^* consisting of the potentially visible portions of the polygons. [Note that, the two supporting lines can be computed using any $O(\log m)$ time sequential algorithm (i.e. [137]), where m is the number of vertices of the convex polygon.]

Stage–2 Let S be the set of straight-line segments joining the two endpoints of the potentially visible chains in R^*.

Stage–3 The lower envelope problem is solved with respect to S and q. Let L be the resulting set of segments.

Stage–4 The left and right endpoints of L are projected (with respect to q) onto R^*.

Stage–5 The non-visible portions of the convex polygons are removed.

The correctness of the algorithm is easy to see.

Theorem 7.2 *There exists a BSP algorithm that, given a set R of n non-overlapping convex polygons in the plane, can determine - with probability at least $1 - O(\frac{1}{n})$ - the visible edges of R in*

$$O(\qquad \max_{k \in [0 \ldots p)} \{ \sum_{i=1}^{\frac{n}{p}} \log m_{ik} \} +$$

$$
\begin{aligned}
&p \ln^2 n \log (p \ln^2 n) + \\
&\frac{n}{p} \log p + \\
&\max \{gp \ln^2 n, L\} + \\
&\log_t p \max \{tgp, L\} + \\
&\max \{g\kappa \frac{n}{p}, L\} + \\
&(p + \kappa \frac{n}{p}) \log (p + \kappa \frac{n}{p}))
\end{aligned}
$$

time, where m_{ik} is the number of vertices of polygon i on processor k, and $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: Obviously, the determination of the potentially visible portions of convex polygons in stage 1 takes time $O(\max_{k \in [0 \ldots p)} \{\sum_{i=1}^{\frac{n}{p}} \log m_{ik}\})$, where m_{ik} is the number of vertices of polygon i on processor k. [Note that, the two supporting lines, delimiting the visible portion, can be computed using any $O(\log m)$ time sequential algorithm.] The computation of the straight-line segments in stage 2 requires $O(\frac{n}{p})$ time steps. The solution of the lower envelope problem in stage 3 requires $O(p \ln^2 n \log (p \ln^2 n) + \frac{n}{p} \log p + \max \{gp \ln^2 n, L\} + \log_t p \max \{tgp, L\} + \max \{g\kappa \frac{n}{p}, L\} + (p + \kappa \frac{n}{p}) \log (p + \kappa \frac{n}{p}))$ time (see theorem 7.1). Finally, stages 4 and 5 take time asymptotically equal to that of stage 1. We thus have the above theorem. ■

It is, therefore, straightforward to verify the following result.

Corollary 7.2 *The visible edges of n non-overlapping convex polygons problem can be solved in*

$$O(\frac{n \log n}{p} + \max_{k \in [0 \ldots p)} \{\sum_{i=1}^{\frac{n}{p}} \log m_{ik}\})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n \log n}{p \log p})$, and $g = O(\log n)$, where m_{ik} is the number of vertices of polygon i on processor k.

Proof: Let in theorem 7.2 $t = 2$. ■

The algorithm that we have presented, for computing the visible edges of non-overlapping convex polygons from a view point, can prove to be useful when the convex polygons (intersections of scan-line planes and three-dimensional objects) are given as part of the input. Such an input is secured in situations when a scene is to be rotated about y-axis or when the same scene is to be reproduced more than once [135].

8 Proximity Problems

Proximity problems arise in a multiplicity of applications where physical or mathematical objects are represented as vectors (points) in space. Thus, for example, in clustering problems it is important to group entities together if they are sufficiently close to each other, in classification it is necessary a new entity to be classified to be assigned to the class of its closest neighbour. Note that, given a set S of n points in E^d, the distance between two points $p : (x_1, x_2, ..., x_n)$ and $q : (x'_1, x'_2, ..., x'_n)$ of S is computed as $(\sum_{i=1}^{d} (x_i - x'_i)^2)^{\frac{1}{2}}$. In the remainder of this section we explore several fundamental proximity problems.

8.1 All-Nearest-Neighbours in the Plane

This problem is so easily stated - given n points in the plane, find the nearest neighbour of each - and important that must be considered as one of the fundamental problems of computational geometry, both from a practical and a theoretical point of view [20]. The *nearest neighbour* is a relation on a set S of n points as follows. Point q is a nearest neighbour of point p if

$$distance(p, q) = \min \{distance(p, z)\}, \text{ for all } z \neq p \in S. \tag{44}$$

Clearly, this relationship is not symmetric. Therefore, if point q is a nearest neighbour of p does not imply that p is a nearest neighbour of q. Furthermore, note that a point is not necessarily the nearest neighbour of a unique point, indeed, a single point might be the nearest neighbour of at most six points, as the angle between those must be at least $\frac{\pi}{3}$.

The central algorithmic issue is whether it is necessary to inspect all $O(n^2)$ pair of points to find the nearest neighbour of each. In [154] it is shown that any sequential algorithm that solves the all-nearest-neighbours problem requires $\Omega(n \log n)$ time steps. This lower bound motivates us to find an $O(\frac{n \log n}{p})$ BSP algorithm for the all-nearest-neighbours problem. Clearly, in one-dimensional Euclidean space an $O(\frac{n \log n}{p})$ algorithm is possible, based on the fact that the nearest neighbours must be consecutive in sorted order [137]. However, since the environment where sorting is useful is a total ordering, sorting is of no use in higher dimensions. Thus, in the remainder of this section, we explore the possibility that some other algorithmic idea (geometric property) might be utilized to improve the brute force algorithm performance.

In [154] a partitioning technique of the two-dimensional Euclidean space in vertical slabs is adopted to handle the sequential version of the all-nearest-neighbours problem. The basic idea behind the sequential algorithm is that the maximum number of points, of separation at least d, that can be packed in a $2d \times d$ rectangle R is at most six. Therefore, at most $O(n)$ distance comparisons were required in the merge step instead of $O(n^2)$. The overall algorithm terminated in optimal $O(n \log n)$ time steps. A similar idea is employed in [55] to solve the parallel version of the problem.

The following lemma derives from the work of [55].

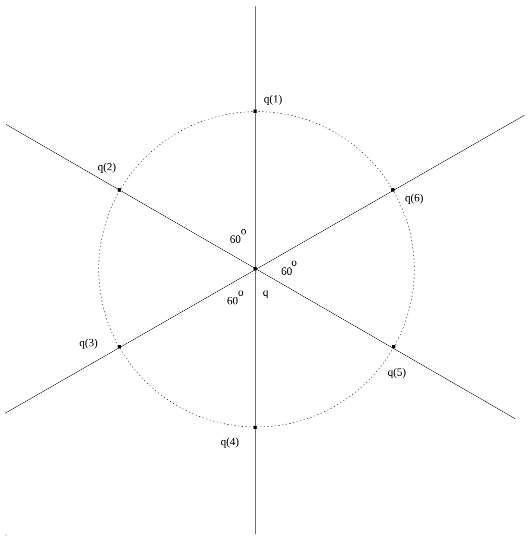

Figure 8: Bounding the number of nearest-neighbours.

Lemma 8.1 *Let V and H be a vertical and a horizontal slab of a point set S. Moreover, let I_{VH} be the set of four intersection points defined by the limiting lines of the two slabs and let R_{VH} be the set of points in $V - H$ whose nearest-neighbour is in I_{VH}. Then, $|R_{VH}| \leq 8$, and for all q in $V - H$ whose nearest neighbour z is in $H - V$ holds that q is in R_{VH}.*

Proof: It is straightforward to verify that the size of I_{VH} is 4 since it results from the pairwise intersection of two vertical and two horizontal lines. Furthermore, any point in the plane can be nearest neighbour to at most 6 points (see figure 8). Similarly, any point in I_{VH} can be the nearest neighbour to at most 2 points in $V - H$. Hence, $|R_{VH}| \leq 8$.

Now we can proceed to the proof of the lemma. Let a point z in $H - V$ be the nearest neighbour to a point q in $V - H$. Assume without loss of generality that q is above H and z is to the right of V. Let r be the upper right intersection of V and H, and refer to figure 9. From Pythagora's theorem it follows that $|\overline{qr}| < |\overline{qo'}|$ and, therefore, that $|\overline{qr}| < |\overline{qz}|$. Since by definition z is the nearest-neighbour of q, it easily follows that q belongs to R_{VH}. ■

It is, therefore, straightforward to verify the following corollary.

Corollary 8.1 *A point p in the intersection of V_i and H_j is the nearest neighbour of either a point in V_i, or in H_i, or in $\cup_i R_{V_i H_j}$.*

Proof: Immediate. ■

Since the algorithm to be presented employs a sequential all-nearest-neighbours algorithm, it is assumed that an optimal $O(n \log n)$ sequential algorithm exists [154].

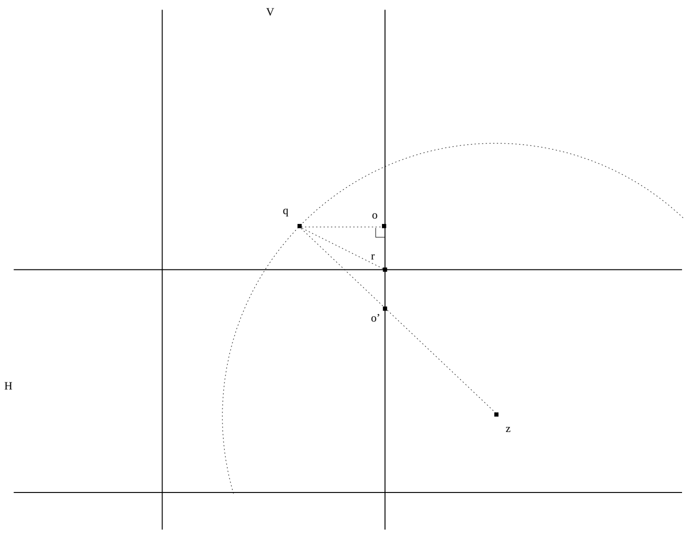

Figure 9: Determining nearest-neighbours within slabs.

Assume that the set S of n points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

The algorithm that follows derives from the *2D-All-Nearest-Neighbours* algorithm of [55] and the above corollary.

Algorithm 8.1

Input: *An array* $S_{i=1\ldots n}$ *of* E^2 *vectors.*
Output: *An array* $S^*_{i=1\ldots n}$*, such that* S^*_i *is the nearest neighbour of* S_i*.*

Stage–1 The bucketing algorithm of section B.2 is used to route the points in S to the appropriate bucket (processor), using as key the x-coordinate, which yields set V_i on processor p_i. [Notice that the resulting buckets are of size $< (1+\frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1-O(\frac{1}{n})$ (see lemmma B.1).]

Stage–2 Each processor p_i solves (sequentially) the all-nearest-neighbours problem for the points in V_i.

Stage–3 The bucketing algorithm of section B.2 is used to route the points in S (along with the nearest neighbour information) to the appropriate bucket (processor), using as key the y-coordinate, which yields set H_j on processor p_j. [Notice that the resulting buckets are of size $< (1+\frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1-O(\frac{1}{n})$ (see lemma B.1).]

Stage–4 A global broadcast operation is initiated, where processor p_j sends line h_j, the horizontal line passing through the point with maximum y-coordinate in H_j, to all other processors.

Stage–5 In order to compute the sets $I_{V_i H_j}$ in p_i, the horizontal lines h_j are merged with the vertical lines delimiting V_i.

Stage–6 Each processor p_i determines the sets $R_{V_i H_j}$, for $0 \leq j < p$. This can be accomplished by sorting V_i locally by y-coordinate and scanning over this ordered sequence.

Stage–7 A global broadcast operation is initiated, where processor p_i sends set $R_{V_i H_j}$ to processor p_j, for $0 \leq j < p$.

Stage–8 Each processor p_j solves (sequentially) the all-nearest-neighbours problem for the points in $H_j + R_{V_i H_j}$, for $0 \leq i < p$.

Stage–9 Clearly, any point q in V_i can have up to p copies in the sets $R_{V_i H_j}$, therefore, it is necessary to route these points back to p_i, in order to determine the nearest neighbour among its copies. Thus, a global broadcast operation is initiated, where processor p_j sends set $R_{V_i H_j}$ (along with the nearest neighbour information) to processor p_i, for $0 \leq i < p$.

Stage–10 Each processor sorts (sequentially) the points received in the previous stage and scans the ordered sequence to determine the copy with the nearest neighbour.

Stage–11 A global broadcast operation is initiated, where every processor sends these unique copies (along with the nearest neighbour information) to the processor matching the proper horizontal slab.

The correctness of the algorithm follows directly from lemma 8.1 and corollary 8.1.

Theorem 8.1 *There exists a BSP algorithm that, given a set S of n points in the plane, can compute the all-nearest-neighbours problem in*

$$
\begin{aligned}
O(\quad & p \ln^2 n \log (p \ln^2 n) + \\
& \frac{n}{p} \log p + \\
& \max \{g p \ln^2 n, L\} + \\
& \log_t p \max \{t g p, L\} + \\
& \max \{g \kappa \frac{n}{p}, L\} + \\
& (p + \kappa \frac{n}{p}) \log (p + \kappa \frac{n}{p}))
\end{aligned}
$$

time - with probability at least $1 - O(\frac{1}{n})$ - where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The bucketing algorithm in stages 1 and 3 takes time $O(p\ln^2 n \log(p\ln^2 n) + \frac{n}{p}\log p + \max\{gp\ln^2 n, L\} + \log_t p \max\{tgp, L\} + \max\{g\kappa\frac{n}{p}, L\})$ (see theorem B.3). The sequential solution of the all-nearest-neighbours problem in stages 2 and the local sorting in stage 6 can be accomplished in $O(\kappa\frac{n}{p}\log(\kappa\frac{n}{p}))$ time steps. The communication complexity in stages 4, 7, 9 and 11 is bounded above by $O(\max\{gp, L\})$. The merging step in stage 5 takes time $O(p)$. The sequential solution of the all-nearest-neighbours problem in stage 8 can be can be computed in $O((p + \kappa\frac{n}{p})\log(p + \kappa\frac{n}{p}))$ time. Finally, the local sorting and scanning in stage 10 requires $O(p\log p)$ time steps. ■

Therefore,

Corollary 8.2 *The all-nearest-neighbour problem can be solved in optimal*

$$O(\frac{n\log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n\log n}{p\log p})$, and $g = O(\log n)$.

Proof: Let in theorem 8.1 $t = 2$. ■

In [154] it is shown that any sequential algorithm that solves the closest pair problem requires $\Omega(n\log n)$ time steps. Therefore, it follows immediately from corollary 8.2 that an optimal $O(\frac{n\log n}{p})$ solution for the closest pair problem can be achieved, on the BSP model, simply by performing a reduce operation (using the binary operator *minimum*) on the output of the all-nearest-neighbours algorithm 8.1.

9 Voronoi Diagrams

At first sight, the following problems, which concern a multiplicity of phenomena at disparate disciplines, would seem to have little in common [128]: the study of the structure of the Universe, the identification of the parts of a region under the influence of different neolithic clans, the estimation of the precipitation at a gauge which has failed to operate, the study of the behaviour of liquid argon, the examination of capillary supply to muscle tissue. However, these phenomena can be dealt with techniques developed from a single concept which forms the subject of this section - the *Voronoi diagram* [33, 41, 67, 70, 94, 99, 100, 113, 114, 136, 152, 153, 169, 171].

The concept is a simple but intuitively appealing one. Given a finite set of distinct points in the plane, we associate all locations in the plane with the closest member of the point set. More formally, given a finite set S of n distinct, isolated points in the plane, for each point z_i in S, determine the locus of points in the plane that are closer to z_i than to any other point of S. We call the z_i's *generator points*. Note that, intuitively, the solution of the above problem is a partitioning of

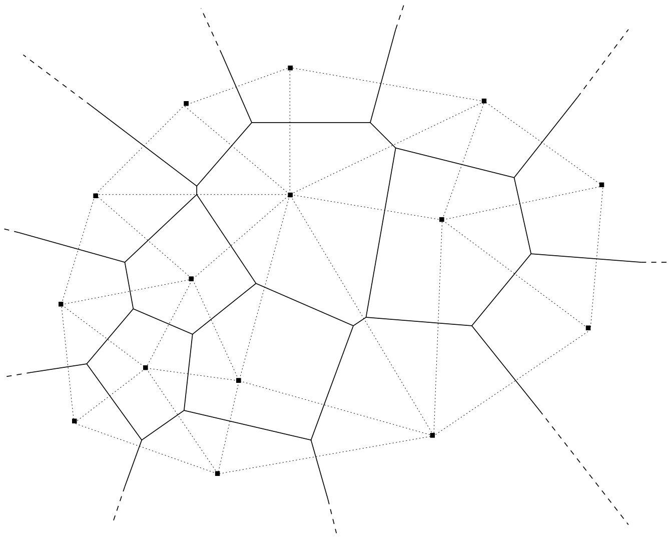

Figure 10: Two-dimensional Voronoi diagram.

the plane into a set of convex regions (probably unbounded). This partitioning is called a *planar ordinary Voronoi diagram*, and the regions constituting the Voronoi diagram are called *ordinary Voronoi polygons*. The boundary of a Voronoi polygon may consist of line segments, half lines or infinite lines, which we call *Voronoi edges*. Moreover, an endpoint of a Voronoi edge is called *Voronoi vertex*. Alternatively, a Voronoi vertex may be defined as a common point shared by three or more Voronoi polygons. When there exists a Voronoi vertex at which at least four Voronoi edges meet in the diagram, we say that the diagram is degenerate [128]. To avoid this complication, we shall make the following assumption. Every Voronoi vertex in a Voronoi diagram has exactly three Voronoi edges as endpoints (degree 3). The solid lines in figure 10 present a simple non-degenerate instance of a Voronoi diagram.

A second diagram (dual) can be constructed from the Voronoi diagram by simply joining those points whose regions share a common edge (see the dotted lines in figure 10). This dual diagram is referred to as the *Delaunay tessellation*. As noted in [128], "Voronoi diagrams and Delaunay tessellations are two of a few truly interdisciplinary concepts with relevant material to be found in, but not limited to, anthropology, archaeology, astronomy, biology, cartography, computational geometry, crystallography, ecology, forestry, geography, geology, marketing, metallography, meteorology, operations research, physics, physiology, statistics and urban and regional planning."

In the following sections we consider practical BSP algorithms for constructing Voronoi diagrams.

9.1 Algorithms for Computing Voronoi Diagrams

As in the previous section, let $S = (z_1, z_2, ..., z_n)$ be a set of n generators, and let $V = (V(z_1), V(z_2), ..., V(z_n))$ denote the Voronoi polygons constituting the Voronoi diagram for S. The construction of the Voronoi diagram is a method for generating V from S [128]. In the preceding section we discussed several definitions for describing the Voronoi diagram; unfortunately, none of these definitions is of a constructive nature. Thus, appropriate notions must be identified that are conductive to algorithm development.

As we have defined it, the Voronoi diagram lives in an unbounded plane. In practical applications, however, it is usual to deal with a bounded region R, where generators are placed. If the region excludes part of the plane, but none of the points in S, then its effect is literally that of blanking out what goes on outside it [85]. We call this set a *bounded Voronoi diagram.*[42] Furthermore, as we observed in figure 10, the Voronoi diagram consists of convex polygons (possibly unbounded). Since a polygon can be defined in terms of half-planes [35], we can alternatively define a Voronoi diagram in terms of half planes.

To illustrate this alternative definition, we consider the line perpendicularly bisecting the line segment $\overline{pq}$ joining two arbitrary generators p and q; we call this line the *bisector* of p and q. Obviously, the bisector divides the plane into two half planes. Assume, without loss of generality, that p is to the left of q, then all the points in the left half-plane are closer to p and all the points in the right half-plane are closer to q. Note that, a point on the bisector is equidistant from the generators p and q. If we call the regions, to the left and right of the bisector, respectively, *dominance region of p over q* and *dominance region of q over p*, it is no surprise that, the Voronoi polygon of a generator point z may be defined as the intersection of the dominance regions of z over all $q \neq z$ in S. Alternatively, the Voronoi polygon $V(z)$ of z can be defined as the intersection of the $n-1$ left half-planes (with reference to a counterclockwise traversal of the boundary of $V(z)$), determined by the $n-1$ bisectors of z in S. Thus we get the following method for computing the Voronoi diagram.

9.1.1 The Naive Method for Constructing Voronoi Diagrams

Although, this method is not recommended for general purposes,[43] it may be used for some special types of simulation where the aim is not to compute the complete Voronoi diagram, but to generate many samples of Voronoi polygons to acquire statistical information about them, i.e. distribution of the number of vertices per polygon [32, 47, 138]. The Voronoi diagram definition of section 9.1, affords us a naive method for constructing the Voronoi diagram. According to this definition we construct the Voronoi polygons one by one, simply by computing the intersection of the $n-1$ half planes defined by the perpendicular bisectors of every generator z_i and the other generators.

[42] Note that, all unbounded Voronoi diagrams can be effectively simulated by sufficiently large bounded ones.

[43] There are known to exist faster methods for constructing the Voronoi diagram.

Let us consider the time complexity of the naive method. The determination of the perpendicular bisector for any two points requires only constant time. Therefore, for each generator z_i the time required for generating the $n-1$ generators is $O(\frac{n}{p})$, where p is the number of processors. Furthermore, to construct the intersection of the $n-1$ half planes, let us consider the following simple method [148]. First, construct the intersection of two half-planes, obtaining a two-sided polygon; next construct the intersection of this polygon with the third plane, and so on. Obviously, this procedure requires time $O(\frac{n^2}{p})$. Since there are n distinct generator points the total time required by the naive method, as described in [148], is $O(\frac{n^3}{p})$. In [128] it is noted that, the time complexity of the naive method can be decreased to $O(\frac{n^2 \log n}{p})$, by employing a more sophisticated technique to construct the intersection of $n-1$ half-planes in $O(\frac{n \log n}{p})$ time. Note that, in the previous analysis we have not included communication and synchronization costs.

In the remainder of this section we shall show how to construct the Voronoi diagram in $O(\frac{n^2}{p})$ time, which is *optimal* for the naive method. Note that, so far, all adequately documented parallel algorithms proposed for this problem use time at least $O(\frac{n^2 \log n}{p})$. Moreover, the algorithm presented here is the first algorithm that is not based on an optimal worst-case algorithm for constructing the intersection of left half-planes.

We shall first show that Euler's formula (see section 2.2.1) implies that the number of vertices (v), edges (e), and faces (f) of a triangulation are linearly related, that is, if $v = n$, then $e = O(n)$ and $f = O(n)$.

Lemma 9.1 *The number of vertices (v), edges (e), and faces (f) of a triangulation are linearly related.*

Proof: If we count the edges of a triangulation face by face, then because each face has three edges, we get $2e = 3f$, since each edge is shared by two faces. Now substitution into Euler's formula establishes the lemma.

$$\begin{aligned} f - e + v &= 2 \\ \frac{2e}{3} - e + v &= 2 \\ e &= 3v - 6 \qquad (45) \\ f &= 2v - 4. \qquad (46) \end{aligned}$$

Therefore, if $v = n$, then $f = O(n)$, $e = O(n)$. ■

Since, the Delaunay tessellation is a triangulation, the above lemma implies that the number of edges of a Voronoi diagram is bounded above by $O(n)$. Note that, there is a one to one correspondence among Voronoi diagram edges and Delaunay triangulation edges (see figure 10).

In [35] it is shown that the n half-planes intersection problem reduces to that of finding the convex hull of n points. Moreover, in section 5.2.1 it is shown that the convex hull of n points can be constructed in $O(\frac{hn}{p})$ time (see corollary 5.1), where h is the expected number of convex hull edges.

Now we can proceed with the Voronoi diagram construction algorithm. Assume that the set S of n generator points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 9.1

Input: *An array* $S_{i=1\ldots n}$ *of* E^2 *vectors.*
Output: *An ordered array* $E^*_{i=1\ldots 3n-6}$*, containing the Voronoi edges of* S*.*

Stage–1 Each processor computes for the points in its possession the $n-1$ bisectors. [Note that this step requires a total of $p-1$ interprocessor messages, each of size $\frac{n}{p}$.]

Stage–2 For each generator point z_i in S all processors cooperate to find the intersection of the $n-1$ left half-planes, determined by the $n-1$ bisectors of z_i. The half-planes intersection can be found using the convex hull algorithm of section 5.2.1.

Stage–3 Stage 2 is repeated until the algorithm determines the entire Voronoi diagram (n iterations).

The correctness of the algorithm is easy to see. The above algorithm affords us with the following theorem.

Theorem 9.1 *There exists a BSP algorithm that, given a set* S *of* n *points in the plane, can construct the Voronoi diagram* $V(S)$ *of* S *in*

$$O(\frac{n^2}{p} + n \log_t p \max\{tg, L\})$$

time.

Proof: The computation in stage 1 takes time $O(n + p \max\{g\frac{n}{p}, L\})$. The convex hull construction is stage 2 requires $O(h_i(\frac{n}{p} + \log_t p \max\{tg, L\}))$ time. Since stage 2 is repeated n times, the overall time complexity of this stage is

$$\begin{aligned} T &= O(\sum_{i=0}^{n-1}(h_i(\frac{n}{p} + \log_t p \max\{tg, L\}))) \\ &= O(\frac{n^2}{p} + n \log_t p \max\{tg, L\}). \end{aligned} \tag{47}$$

Note that, $\sum_{i=0}^{n-1} h_i = O(n)$ (see lemma 9.1). We thus obtain the above theorem. ∎

From theorem 9.1 it follows directly that,

Corollary 9.1 *The Voronoi diagram problem can be solved in worst-case time*

$$O(\frac{n^2}{p}),$$

on the BSP model, for any integers n, p, L, *and* g *such that* $L = O(\frac{n}{p \log_t p})$, *and* $g = O(\frac{n}{tp \log_t p})$.

Proof: Immediate. ■

Therefore, the asymptotic time complexity of the naive method algorithm 9.1, presented in this section, compares to that of far more sophisticated algorithms, i.e. "incremental" method algorithm, "walking method" algorithm, and "modification method" algorithm. In the following section, a straightforward but still very powerful method, for constructing the Voronoi diagram in $O(g\frac{n}{p})$ expected time is presented.

9.1.2 An Optimal Expected Time Algorithm for Constructing Voronoi Diagrams

In this section we shall show how the Voronoi diagram of n points drawn independently from a uniform distribution function in the unit square can be constructed in $O(g\frac{n}{p})$ expected time.[44] The algorithm uses the technique of grid-partitioning previously used in [24], appropriately modified to perform efficiently in a parallel environment.

We shall now briefly sketch the algorithm before examining it in detail. Let $S = (z_1, z_2, ..., z_n)$ be a set of n generator points uniformly distributed over $[0,1]^2$ (unit square). The first step is to partition S in p vertical slabs such that each slab contains $O(\frac{n}{p})$ points. Then $[0,1]^2$ is divided into n cells, each with edge length $n^{-\frac{1}{2}}$, and each point is assigned to the appropriate cell. [Note that, assigning each point to the appropriate cell, can be accomplished in linear time since we have assumed that the computation of the floor function takes constant time (see section 2.1.3).] The next step is to construct separately the Voronoi polygons of the points in the *inner* and *outer* sets, the outer cells containing all cells "near" the periphery of $[0,1]^2$. This distinction is essential since points near the periphery of a Voronoi diagram do not exhibit locality [24]. A cell-based constant time algorithm is then applied to every point in the inner cells, and a worst-case algorithm is applied to the (small) set of points in the outer cells. The expected running time of both algorithms is at most linear. The algorithm will produce, for each point, a linked list of all its Voronoi neighbours, along with a representation of the line segment that is the Voronoi edge separating the two points. This representation is very attractive, since one can convert from it to any other proposed representation in linear expected time [24]. A rather natural data structure, that embodies most of the properties of a grid, proves to be the two-dimensional array as rendered in figure 11. To access the points in a particular cell $[i,j]$, one simply reads the head

[44] In the remainder of this section we shall assume that the input points are drawn independently from a uniform distribution in $[0,1]^2$.

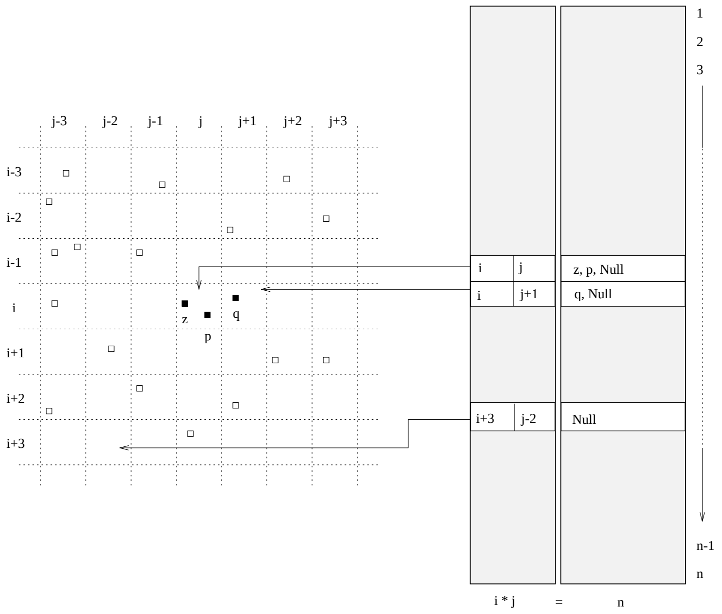

Figure 11: Cell data structure.

of the linked list at address $j + \sqrt{n}i$, and then, sequentially accesses the remaining points.

This data structure is presented in some detail because it is thought to be of general interest for any algorithm using some kind of grid. Furthermore, it can be easily extended to higher dimensions; in the three-dimensional case one first partitions on z-coordinates and then for every $x-, y-$plane partitions as in figure 11.

Now we can proceed with the Voronoi diagram algorithm. Assume that the set S of n points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 9.2

Input: *An array $S_{i=1\ldots n}$ of E^2 vectors (uniformly distributed over $[0,1]^2$.*
Output: *An ordered array $E^*_{i=1\ldots 3n-6}$, containing the Voronoi edges of S.*

Stage–1 The bucketing algorithm of section B.2 is used to route the points in S to the appropriate bucket (processor), using as key the x-coordinate, which yields vertical slab V_i on processor p_i. [Notice that the resulting buckets are of size $< (1+\frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–2 Each processor then partitions the set S into n equally sized cells, each with edge length $n^{-\frac{1}{2}}$.

Stage–3 Each processor determines the cell to which each point in its possession belongs. [Note that the floor function will be employed in this step, i.e. multiply both coordinates of the point by $\sqrt{n}$ and apply the floor function to determine the x- and y-position of the cell to which the point belongs.]

Stage–4 All cells no further than $\log n$ layers[45] from the periphery of the square are defined to be outer cells; all remaining cells are defined to be inner cells.

Stage–5 The Voronoi polygon of each point in the inner cells is constructed in constant expected time. [Note that, to find the Voronoi polygon for some inner point z_i, we search, in a spiral fashion (see figure 12), all cells in the neighbourhood of z_i until at least one point is found in each of the six sextans shown in figure 13, or we quit having examined at most $\log n$ layers in each sextan. Then we proceed (in a spiral fashion) by examining (on the average) a constant number of additional cells (see lemma 9.2) to correctly compute the Voronoi polygon of z_i. As it is shown in lemma 9.2, the total number of cells and points that have to be considered in the computation of the Voronoi polygon of z_i is constant (on the average). Finally, note that for points near the boundary of V_i some interprocessor communication is necessary.]

Stage–6 All cells no further than $\sqrt{\frac{16}{3}}\log n$ layers from the outer cells are defined to be *middle* cells. The Voronoi diagram of the union of outer and middle cells is constructed using some worst-case algorithm. [Note that, the number of outer and middle cells is $4(\log n + \sqrt{\frac{16}{3}}\log n)\sqrt{n} - 4(\log n + \sqrt{\frac{16}{3}}\log n)^2$, which is also an upper bound on the expected number of points in those cells.

We shall now show that with very high probability we have correctly computed the Voronoi polygon of each point, and that if we have not, then the situation can be quickly detected and corrected.

Stages 1, 2, 3 and 4 are trivial. The correctness of the stage 5 needs to be established on a twofold basis. First, we have to prove that the spiral cell method is indeed a constant time solution to the Voronoi polygon construction problem, and second, we have to ascertain that by examining at most $\log n$ layers in each sextan we locate (with high probability) at least one point in each of them. The constant time property of the spiral cell method is established in lemma 9.2.

We shall now prove that by inspecting $\log n$ layers in each sextan we find with high probability at least one point in each of them. Since the number of cells in the grid structure is n, the probability of a generator point being placed in a particular fixed cell is $\frac{1}{n}$. Similarly, the probability that a generator point is not placed in a

[45] We say that the cell containing z_i is in layer 1, the 8 cells surrounding z_i are in layer 2, the 16 cells surrounding those are in layer 3, and so on. Therefore, for $l > 1$, there are exactly $8(l-1)$ cells on the l-th layer.

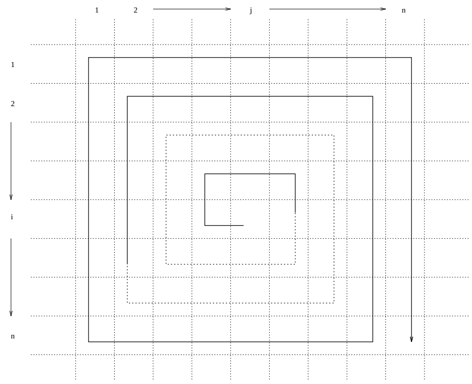

Figure 12: Spiral-search method.

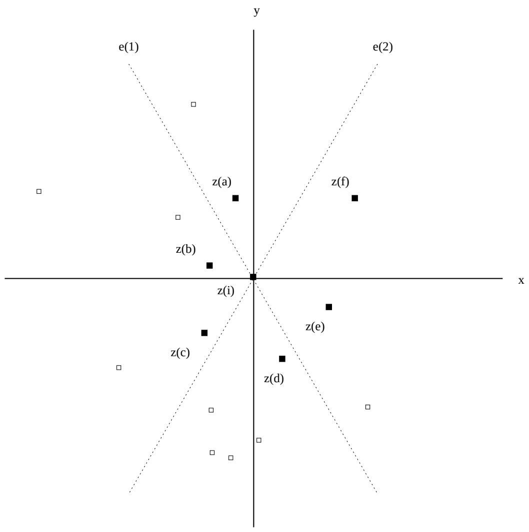

Figure 13: Point location in sextans.

particular fixed cell is $1 - \frac{1}{n}$, and, therefore, the probability that a single particular cell is empty is,

$$(1 - \frac{1}{n})^n \leq e^{-1}, \tag{48}$$

since it is empty if and only if all n points are placed elsewhere. Accordingly, the probability that m given cells are all empty is bounded above by e^{-m}. Furthermore, the number of cells on or within layer l is $(2l-1)^2$ (see lemma 9.2). Thus, the probability that a point is not found within $\log n$ layers (in each sextan) (roughly $\frac{2}{3}\log^2 n$ cells) is bounded above by,

$$6e^{-\frac{2}{3}\log^2 n} = 6n^{-\frac{2}{3\ln 2}\log n}. \tag{49}$$

Since the number of inner points is less than n, the probability that stage 5 fails for one or more of them is at most,

$$6nn^{-\frac{2}{3\ln 2}\log n} = 6n^{1-\frac{2}{3\ln 2}\log n}. \tag{50}$$

Thus, we see that by searching in a spiral fashion the cells in the neighbourhood of a generator point z_i we find at least one point in each of the six sextans shown in figure 13 with high probability. As we have shown, the probability of failure is so small that it doesn't seem worthwhile to invest much effort to treat it. For instance, a trivial solution would be that of employing a deterministic sequential algorithm (implemented by message passing) whenever stage 5 fails.

Lemma 9.2 *The spiral cell method constructs the Voronoi polygon of a generator point z_i in constant expected time, under the assumption that the generator points $S = (z_1, z_2, ..., z_n)$ are drawn independently from a uniform distribution function in the unit square.*

Proof: Let us define certain notions that will be commonly used in the remainder of this proof. We first define the notion of *layers* of cells surrounding a generator point z. We say that the cell containing z is in layer 1, the 8 cells surrounding z are in layer 2, the 16 cells surrounding those are in layer 3, and so on. Therefore, for $l > 1$, there are exactly $8(l-1)$ cells on the l-th layer. Therefore, the number of cells on or within layer l is bounded above by,

$$1 + \sum_{i=2}^{l} 8(i-1) = 1 + 8\frac{l(l-1)}{2} = (2l-1)^2 < 4l^2. \tag{51}$$

Since the number of cells in the grid structure is n, the probability of a generator point being placed in a particular fixed cell is $\frac{1}{n}$. Similarly, the probability that a generator point is not placed in a particular fixed cell is $1 - \frac{1}{n}$, and, therefore, the probability $P^{(1)}_{empty}$ that a single particular cell is empty is,

$$\begin{aligned} P^{(1)}_{empty} &= (1 - \frac{1}{n})^n \\ &\leq e^{-1}, \end{aligned} \tag{52}$$

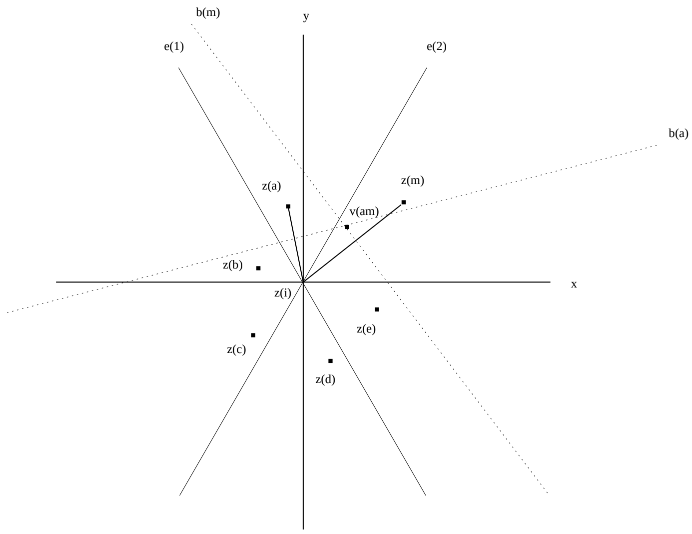

Figure 14: Bounding the distance within which Voronoi vertices are expected to be located.

since it is empty if and only if all n points are placed elsewhere. Accordingly, the probability $P_{empty}^{(m)}$ that m given cells are all empty is bounded above by e^{-m}. Now we can proceed to the proof of lemma 9.2.

Let m be the maximum number of cells searched in any of the six sextans before a point was found in them, and let $V'(z_i)$ be the Voronoi polygon of z_i with respect to those six points. Furthermore, let d_{z_m} be the distance from z_i (the generator point) to the furthest point z_m and refer to figure 14. Assume, without loss of generality, that z_m lies in the sextan defined by the half-lines $z_i x$ and $z_i e_2$. Moreover, let z_a be the point found in the sextan defined by the half-lines $z_i e_1$ and $z_i e_2$. Construct the perpendicular bisectors b_m and b_a of $\overline{z_i z_m}$ and $\overline{z_i z_a}$ respectively, and let v_{am} be their intersection. Note that v_{am} is a potential vertex of $V(z_i)$. It is not difficult to see that, the distance $d_{v_{am}}$ between z_i and v_{am} increases proportionally to the angle $\angle z_a z_i z_m$, and assumes its maximum value when $\angle z_a z_i z_m = \frac{2\pi}{3}$ and $|\overline{z_i z_a}| = |\overline{z_i z_m}|$ (see figure 15). Therefore,

$$\begin{aligned} \frac{d_{z_m}}{2} &\leq \sin\left(\frac{\pi}{6}\right)\max\{d_{v_{am}}\} \\ d_{z_m} &\leq \max\{d_{v_{am}}\}. \end{aligned} \tag{53}$$

Note that, points in non-consecutive sextans can not affect $\max\{d_{v_{am}}\}$. Then, no point further than $2d_{z_m}$ from z_i can have any effect on the actual Voronoi polygon of z_i, which implies that $V(z_i)$ can be constructed by considering only $\sqrt{\frac{16}{3}}l_m$ layers (refer to figure 16), where l_m is the layer on which z_m lies.[46] We now

[46]Consider the triangle $\triangle z_m z_i z'_m$, where z'_m is the projection of z_m onto the y-axis, and refer

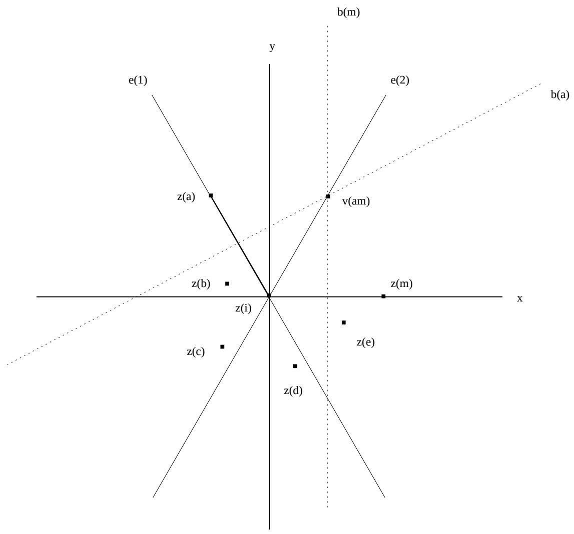

Figure 15: Bounding the maximum distance within which Voronoi vertices are expected to be located.

use equation 51 to upper bound $\sqrt{\frac{16}{3}} l_m$.

$$\begin{aligned} m &> \frac{(2(l_m-1)-1)^2}{6} \\ 2l_m &< \sqrt{6m}+3 \\ \sqrt{\frac{16}{3}} l_m &< \sqrt{\frac{4}{3}}(\sqrt{6m}+3). \end{aligned} \tag{54}$$

Substituting the above result in equation 51 we obtain an upper bound on the number of cells c_{max} to be considered.

$$\begin{aligned} c_{max} &< \frac{4(\sqrt{\frac{16}{3}} l_m)^2}{6} \\ &< \frac{4(\sqrt{\frac{4}{3}}(\sqrt{6m}+3))^2}{6} \\ &= \frac{8}{9}(6m+6\sqrt{6m}+9) \\ &\leq \frac{8}{9}(21m+9) \\ &\leq 19m+8. \end{aligned} \tag{55}$$

to figure 16. Since the angle $\angle z_m z_i z'_m = \frac{\pi}{6}$, it is easy to verify that $\overline{z_i z_m} = 2\overline{z_m z'_m}$. Then, from Pythagoras' theorem we have that $2|\overline{z_i z_m}| = \frac{4}{\sqrt{3}} l_m$.

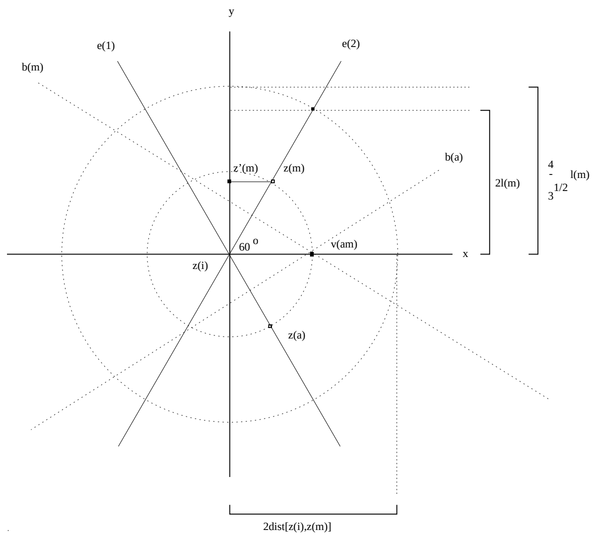

Figure 16: Maximum number of layers to be searched using the spiral-search method to determine a Voronoi polygon.

Hence, the total number of cells one need search is at most $19m+8$, or $18m+8$ beyond the original m visited.

Let $E(m)$ denote the expected number of generator points need to be inspected by the spiral method when the first point is found in cell m. Decompose now $E(m)$ into $E(m) = E_f(m) + E_r(m)$, where $E_f(m)$ is the expected number of points in the first non-empty cell (the m-th cell), and $E_r(m)$ is the expected number of points in the remaining $\leq 18m+8$ cells.

Letting p_i denote the probability of observing i points in a certain fixed cell if n points are distributed randomly in $(n-m+1)$ cells, and $q = \frac{1}{n-m+1}$ denote the probability of a point being placed in a certain fixed cell, we have,

$$\begin{aligned}
E_f(m) &= \frac{\sum_{i=1}^{n} i p_i}{\sum_{i=1}^{n} p_i} \\
&= \frac{\sum_{i=1}^{n} i \begin{pmatrix} n \\ i \end{pmatrix} q^i (1-q)^{n-i}}{1-p_0} \\
&= \frac{nq}{1-(1-q)^n} \\
&= \frac{\frac{n}{n-m+1}}{1-(\frac{n-m}{n-m+1})^n} \\
&\leq \frac{\frac{n}{n-m+1}}{1-(\frac{n-m}{n-m+1})^{n-m+1}}
\end{aligned}$$

$$
\begin{aligned}
&\leq \frac{\frac{n}{n-m+1}}{1-e^{-1}} \\
&< \frac{2n}{n-m+1}. \qquad (56)
\end{aligned}
$$

To evaluate $E_r(m)$, we let i denote the number of points found in cell m; therefore, we have,

$$
\begin{aligned}
E_r(m) &\leq \max_{i=1\ldots n} \{\frac{(18m+8)(n-i)}{n-m}\} \\
&\leq \frac{(18m+8)(n-1)}{n-m}. \qquad (57)
\end{aligned}
$$

Hence, $E(m)$ is bounded above by (see equations 56 and 57),

$$
\begin{aligned}
E(m) &= E_f(m) + E_f(m) \\
&\leq \frac{2n}{n-m+1} + \frac{(18m+8)(n-1)}{n-m} \\
&\leq 2(18m+10), \text{ for all } m \leq \frac{n}{2}. \qquad (58)
\end{aligned}
$$

Therefore,

$$
E(m) = \begin{cases} 2(18m+10), & \text{if } m \leq \frac{n}{2} \\ n, & \text{otherwise.} \end{cases}
$$

The expected cost contributed by the m-th cell is the sum of the number of cells examined and the expected number of points in those cells, times the probability of its being the first non-empty - which is bounded above by e^{1-m} (see equation 52):

$$
e^{1-m}(19m + 8 + E(m)). \qquad (59)
$$

Thus, the total expected cost $T(n)$ of the spiral method for constructing the Voronoi polygon of an arbitrary generator point z, is simply this product summed over all values of m from 1 to n.

$$
\begin{aligned}
T(n) &\leq \sum_{m=1\ldots\frac{n}{2}} e^{1-m}(19m + 10 + 2(18m+10)) + \sum_{m=(\frac{n}{2}+1)\ldots n} e^{1-m}n \\
&\leq \sum_{m=1\ldots\frac{n}{2}} e^{1-m}O(m) + O(1) \sum_{m=\frac{n}{2}\ldots n} e^{-\frac{n}{2}}n \\
&\leq O(1) \sum_{m=1\ldots\frac{n}{2}} e^{-m}O(m) + O(1) \sum_{m=1\ldots\frac{n}{2}} e^{-\frac{n}{2}}n \\
&\leq O(1) + O(e^{-\frac{n}{2}}n^2) \\
&= O(1). \qquad (60)
\end{aligned}
$$

[Note that, $\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$, for any $|x| < 1$ [68].]

We have therefore established the above lemma. [Note that, in the above analysis we have considered the "worst-case" sextan - the one with the furthest point. It is therefore easy to see that the above-stated lemma generalizes to all six sextans.] ■

In the analysis of stage 5 we have deliberately overlooked the fact that for points near the processor-grid boundaries some interprocessor communication is required. Evidently, since by lemma 9.2 no processor will search further than $\sqrt{\frac{16}{3}} \log n$ layers, no more than $\sqrt{\frac{16}{3} n} \log n$ cells (points) need to be communicated, which is also an upper bound on the expected number of messages to be sent or received. This result concludes the analysis of stage 5.

We shall now show that stage 6 correctly computes the Voronoi polygon of each outer point. Since by lemma 9.2 the Voronoi polygon of inner points cannot be affected by points further than $\sqrt{\frac{16}{3}} \log n$ cells, no inner cell (note that, inner cells should now be distinguished from middle cells) shares a common edge with an outer cell, and, therefore, stage 6 correctly computes the Voronoi polygon of outer points.

The above analysis affords us with the following theorem.

Theorem 9.2 *There exists a BSP algorithm that, given a set S of n generator points in the plane, can compute the Voronoi diagram $V(S)$ of S in*

$$
\begin{aligned}
O(\quad & p \ln^2 n \log (p \ln^2 n) + \\
& \kappa \frac{n}{p} \log p + \\
& \max \{gp \ln^2 n, L\} + \\
& \log_t p \max \{tgp, L\} + \\
& \max \{g\kappa \frac{n}{p}, L\} + \\
& \max \{g\sqrt{n} \log n, L\})
\end{aligned}
$$

expected time, where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The bucketing algorithm in stage 1 takes time $O(p \ln^2 n \log (p \ln^2 n) + \kappa \frac{n}{p} \log p + \max \{gp \ln^2 n, L\} + \log_t p \max \{tgp, L\} + \max \{g\kappa \frac{n}{p}, L\})$ (see theorem B.3). Obviously, the local computations in stages 2, 3, 4 and 5 require $O(\frac{n}{p})$ time steps. As we have seen, the communication requirement of stage 5 is bounded above by $O(\max \{g\sqrt{n} \log n, L\})$. Finally, stage 6 takes $O(\frac{(\sqrt{n} \log n - \log^2 n) \log (\sqrt{n} \log n - \log^2 n)}{p}) \leq O(\frac{n}{p})$ time[47]. ■

Therefore,

[47]The assumption is here that an $O(\frac{n \log n}{p})$ worst-case parallel Voronoi diagram algorithm exists; in fact, any worst-case polynomial-time $O(n^{\epsilon})$ algorithm suffices, provided that $0 \leq \epsilon < 2$.

Corollary 9.2 *The Voronoi diagram problem can be solved in*

$$O(g\frac{n}{p})$$

expected time on the BSP model, for any integers n, p, L, *and* g *such that* $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n \log n}{p \log p})$, *and* $g = \Omega(\log(p \ln^2 n))$.

Proof: Let in theorem 9.2 $t = 2$. ■

Undoubtedly, for all large enough n, the running time of algorithm 9.2 is dominated by the bucketing operation in stage 3.

Although the analysis of the above algorithm is rather tedious, the algorithm itself can be easily comprehended on an intuitive level. Stated very briefly, Voronoi diagrams are local phenomena, and so are cells. As noted in [24],

> ... Suppose you are standing in the middle of a large hall that is covered with tiles that are 1 foot by 1 foot square; suppose furthermore that the hall has been sprinkled uniformly with pennies, so that there are a dozen pennies per tile, on the average. How many feet out will you have to look before you find the penny nearest you? Your answer will be independent of the size of the hall, because the density of pennies is the critical issue, and not their absolute number; whether the hall is 100 feet square or 1 mile square is immaterial!

10 Geometric Optimization

In this section we discuss problems that ask for the construction of an Euclidean geometric object minimizing (or respectively maximizing) a given objective function. We select for presentation the problem of computing minimum-area enclosing rectangles.

10.1 Minimum-area Enclosing Rectangle

This section describes a method for identifying the minimum-area rectangle in which a given set of points in the plane can be contained. More formally, given a set $S = (v_1, v_2, ..., v_n)$ of n points in the plane, find the rectangle of minimum-area that encloses them. This problem occurs frequently in a variety of industrial applications, in particular, those related to optimum two-dimensional layout [2, 86].

Freeman [73], provided a sequential algorithm for solving the minimum-area enclosing rectangle problem in $O(n^2)$ time. In this report we improve the efficiency of the sequential algorithm by presenting an $O(\frac{n \log n}{p})$ time BSP algorithm. Moreover, conceptually, the new algorithm is even simpler than that of [73].

To obtain the desired time complexity for the minimum-area enclosing rectangle problem, we shall make use of the following results.[48]

[48]Note that, while the area of this rectangle is unique, the rectangle itself might not be.

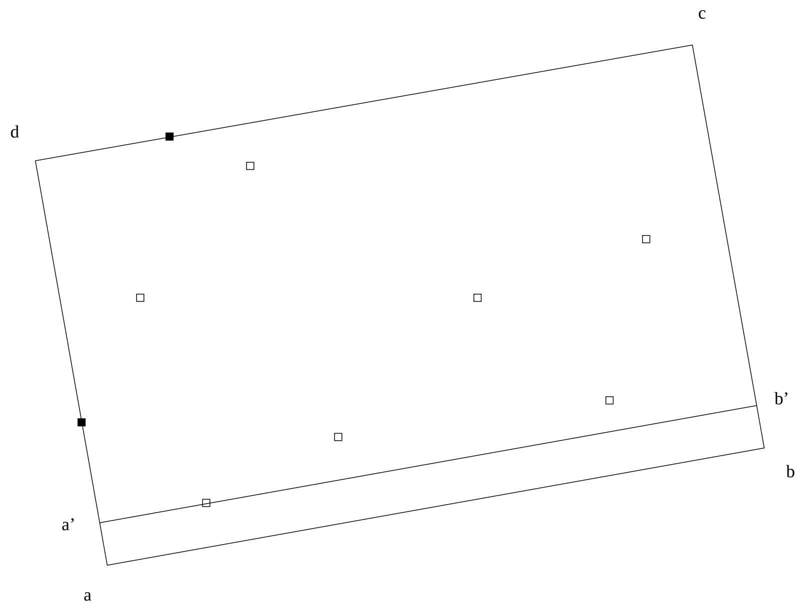

Figure 17: Determining the minimum-area enclosing rectangle.

Lemma 10.1 *Given a set of points S and an enclosing rectangle R of S, such that $k < 4$ edges of R contain a point of S, there exists another rectangle R' such that $k+1$ edges of R' contain a point of S and the area of R' is less than that of R.*

Proof: Select a rectangle R with exactly $k < 4$ edges containing at least one point of S (and accordingly 4 - k edges containing no points of S), and refer to figure 17. Without loss of generality, assume that the edge $\overline{ab}$ contains no points of S. We now construct another rectangle R' with the same sides and such that $\overline{ab}$ has been shifted appropriately ($\overline{a'b'}$) to contain a point in S. Obviously, there are $k+1$ edges of R' that contain a point of S.

Let Δx be the vertical distance between $\overline{ab}$ and $\overline{a'b'}$, and let ΔA be the amount by which the area of R exceeds that of R'. Using the notation of figure 17, we obtain,

$$\Delta A = \Delta x |\overline{ab}|, \tag{61}$$

which is positive for all $\Delta x > 0$, $|\overline{ab}| > 0$. Obviously, $|\overline{ab}| > 0$ since by definition a rectangle has positive length sides. It remains to prove that $\Delta x > 0$. Suppose not, i.e. suppose $\Delta x = 0$ (Δx cannot be negative); then, $\overline{ab} = \overline{a'b'}$ which cannot be true since by definition $\overline{ab}$ contains no points of S and $\overline{a'b'}$ contains at least one point of S. Thus, the lemma follows. ■

Corollary 10.1 *The minimum-area enclosing rectangle R of a set of points S contains at least one point of the convex hull of S in each of its sides.*

Proof: Suppose not, i.e. suppose there exists a minimum-area enclosing rectangle R such that (a) one of its edges contains no points of S, or (b) one of its edges contains a non-convex hull point of S.

[**Case a**] If the minimum-area enclosing rectangle R of S has an edge that contains no points of S, then, by lemma 10.1, there exists some enclosing rectangle R' such that the area of R' is less than that of R, and this contradicts the minimum-area property of R.

[**Case b**] Suppose that the minimum-area enclosing rectangle R of S has an edge that contains a non-convex hull point q of S. Then, by the definition of the enclosing rectangle, the edge of R passing through q has all the points of S on one and same side of it, and this contradicts the convex hull definition (see section 2.2.1).

In either case, the corollary is proved. ■

The following was observed in [73].

Lemma 10.2 *The minimum-area enclosing rectangle of a set of points S has a side collinear with one of the edges of the convex hull of S.*

Proof: A detailed account of the above result is given in [73]. ■

The above results afford us with a naive $O(\frac{n^2}{p})$ time algorithm for the minimum-area enclosing rectangle problem, i.e., (a) find the convex hull $CH(S)$ of S, (b) for each edge e_i in $CH(S)$ compute the minimum-area enclosing rectangle[49] that has a side collinear with e_i, (c) select the rectangle of minimum area out of those generated in the previous step. Indeed, the above is an outline of the algorithm presented in [73].

Although activities (a) and (c) require careful implementation, it is clear that activity (b) is the crux (and bottleneck) of the method. Indeed, one may instinctively fear that it may require work proportional to $\frac{n^2}{p}$, corresponding to determining the minimum-area enclosing rectangle for each convex hull edge. Fortunately, the machinery of parallel all-prefix comes to our rescue with the following lemma.

Lemma 10.3 *Once the convex hull edges have been sorted by slope, the minimum-area enclosing rectangle corresponding to each convex hull edge can be found for all convex hull edges simultaneously by performing a single parallel all-prefix operation.*

Proof: Earlier in this section we demonstrated how to find the minimum-area enclosing rectangle in time $\Theta(1)$, once we are given an edge $e = \overline{qq'}$ collinear to one

[49] Given an edge e with endpoints q and q' the minimum-area enclosing rectangle can be determined as follows. Set up an $x'y'$ coordinate system with origin at q and positive x'-axis directed toward q'. Compute the $x'y'$ coordinates for all the vertices in the convex hull vertex list, and search the updated vertex list for the points, $\min\{x'\}$, $\max\{x'\}$, and $\max\{y'\}$. In constant time compute the minimum-area enclosing rectangle defined by its edge $\overline{qq'}$ and the three points $(\min\{x'\}, \max\{x'\}, \max\{y'\})$.

of its sides and the set of points q_E, q_W, q_N that satisfy, respectively, the functions $\min\{x'\}$, $\max\{x'\}$, and $\max\{y'\}$, where $x'y'$ is the coordinate system with origin at q and positive x'-axis directed toward q'. Hence, to find the minimum-area enclosing rectangle corresponding to each convex hull, it is sufficient to compute for each edge the three points q_E, q_W, q_N. In the remainder of this section we shall concentrate in computing q_N; q_E and q_N can be found in a similar fashion simply by rotating the coordinate system by $\pm\frac{\pi}{2}$.

Consider the straight line segment $\overline{q_{x-min}q_{x-max}}$, where q_{x-min} and q_{x-max} are, respectively the convex hull points with minimum and maximum x-coordinates. Clearly, the points q_{x-min} and q_{x-max} partition the convex hull CH into two halves (not necessarily equal). Let UH (Upper Hull) be the partition of CH consisting of all points from q_{x-min} to q_{x-max} (clockwise), and LH (Lower Hull), defined similarly from q_{x-max} to q_{x-min}. Moreover, let e_l and e_r denote, respectively, the edges to the left and right of q_N (note that, q_N is the right (left) endpoint of e_l (e_r)).

It is not difficult to verify that,

$$slope(e_l) \leq slope(e) \leq slope(e_r), \tag{62}$$

where $e = \overline{qq'}$. Furthermore, notice that for every edge e in LH, q_N belongs to UH, and, by symmetry, for every edge e in UH, q_N belongs to LH. The proof is as follows. Suppose not, i.e., suppose that for some edge e in LH, q_N belongs to LH, then, there must exist two edges e_l and e_r in LH such that inequality 62 is satisfied. Obviously, the only pair of edges in LH that satisfy inequality 62 is the pair of edges directly to the left and right of e, and, therefore, q_N has to be an endpoint of e, which is invalid.

Now we are ready to establish the main result of lemma 10.3. Consider the set of edges sorted by slope and labelled as upper and lower hull edges. Then, from inequality 62 and the discussion of the previous paragraph, we have that, for each edge e in LH, q_N is the right endpoint of an edge e_{UH} in UH with the largest slope, such that $slope(e_{UH}) < slope(e)$; by symmetry, for each edge e in UH, q_N is the right endpoint of the edge e_{LH} in LH with the largest slope, such that $slope(e_{LH}) < slope(e)$. Since the set of edges is pre-sorted by slope, it is sufficient to perform an all-prefix operation on the slopes, using the binary operator *maximum**, to compute q_N for all edges in CH. The binary operator *maximum** is the ordinary *maximum* operator, slightly extended so that, for an edge e in LH returns the maximum slope edge e_{UH} in UH such that $slope(e_{UH}) < slope(e)$, and for an edge e in UH returns the maximum slope edge e_{LH} in LH such that $slope(e_{LH}) < slope(e)$. We thus have the above lemma. ■

The details of the algorithm are as follows. Assume that the set S of n points is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm 10.1

Input: *An array $S_{i=1...n}$ of E^2 vectors.*

Output: *The minimum-area enclosing rectangle.*

Stage–1 The algorithm of section 5.2.2 is used to compute the convex hull $CH(S)$ of S.

Stage–2 A reduce operation is initiated, using the binary operators *minimum* and *maximum*, to determine the convex hull points q_{x-min} and q_{x-max} with minimum and maximum x-coordinates respectively.

Stage–3 Each processor labels the $CH(S)$ edges in its possession as either *lower* or *upper* hull edges, depending on whether they are, respectively, below or above the straight line segment $\overline{q_{x-min}q_{x-max}}$.

Stage–4 The algorithm of section B.3 is used to sort the $CH(S)$ edges in ascending order, using as key the edge slopes. [Note that, edges can be sorted, without actually computing any slopes, but, rather, by employing cross product properties.]

Stage–5 An all-prefix operation is initiated, using the binary operator *maximum** (see lemma 10.3), to determine the three points q_E, q_W, and q_N for each edge in $CH(S)$.

Stage–6 Each processor computes, for each edge in its possession, the minimum-area enclosing rectangle in $\Theta(1)$ time.

Stage–7 A reduce operation is initiated, using the binary operator *minimum*, to determine the rectangle of minimum area out of those generated in the previous stage.

The correctness of the algorithm follows directly from lemma 10.3, and corollary 10.1.

Theorem 10.1 *There exists a BSP algorithm that, given a set S of n points in the plane, can compute the minimum-area enclosing rectangle of S in*

$$
\begin{aligned}
O(\quad & p\ln^2 n\log(p\ln^2 n) + \\
& \kappa\frac{n}{p}\log p + \\
& \kappa\frac{n}{p}\log(\kappa\frac{n}{p}) + \\
& \max\{gp\ln^2 n, L\} + \\
& \log_t p\max\{tgp, L\} + \\
& \max\{g\kappa\frac{n}{p}, L\} + \\
& \log(\kappa\frac{n}{p})\max\{pg, L\})
\end{aligned}
$$

time - with probability at least $1 - O(\frac{1}{n})$ - where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: The convex hull computation (see theorem 5.2) takes $O(p\ln^2 n \log{(p\ln^2 n)} + \kappa\frac{n}{p}\log p + \kappa\frac{n}{p}\log{(\kappa\frac{n}{p})} + \max{\{gp\ln^2 n, L\}} + \log_t p \max{\{tgp, L\}} + \max{\{g\kappa\frac{n}{p}, L\}} + \log{(\kappa\frac{n}{p})}\max{\{pg, L\}})$ time, and, clearly, dominates the time complexity of the algorithm. ■

Accordingly,

Corollary 10.2 *The minimum-area enclosing rectangle problem can be solved in*

$$O(\frac{n\log n}{p})$$

time , on the BSP model, for any integers n*,* p*,* L*, and* g *such that* $p = O(\frac{\sqrt{n}}{\log n})$*,* $L = O(\frac{n\log n}{p\log p})$*, and* $g = O(\log n)$*.*

Proof: Let in theorem 10.1 $t = 2$. ■

Note that, the probabilistic feature of the algorithm can be eliminated if the sorting algorithm of section B.3 is replaced by a similar deterministic algorithm (i.e., see algorithm B.4).

11 Future Directions

As its title indicates, this final chapter aspires to point toward some directions for future research in BSP computational algorithms. We discuss implementing data structures on BSP machines, improving current and future algorithms as to require less parallel slackness, modifying existing expected-time algorithms to high-probability ones, studying problems related to dynamic computational geometry, geometric searching, geometric optimization, maxima, intersections, convex hulls, triangulations, arrangements, proximity, separability, isothetic geometry, and visibility. Moreover, we would be interested in extending our investigation of BSP algorithms to other algorithmic areas, such as graph, string and pattern matching algorithms.

11.1 Implementing Data Structures on BSP machines

In theoretical computer science, parallel computation has been traditionally studied in a model of parallel access machine called the PRAM. The most important feature of the PRAM model, and the reason for its power, is the common memory shared by the processors [13]: "...not only does the shared memory serve as a communication medium for the processors, but it allows a direct implementation of complex data structures, in a manner very similar to the way they are implemented on the memory of a sequential computer." Unfortunately, the situation is considerably different for BSP machines, where the memory is no longer considered as shared but instead, distributed among the processors. Implementing data structures in the latter situation offers a more serious challenge [13].

The problem of implementing such distributed data structures has been addressed in the past and several promising results have been obtained for particular network architectures, i.e., hypercubes, meshes [54]. Unfortunately, due to the architecture dependent features involved, it remains open whether these results can be migrated to the more general setting of the BSP model. These elements suggest that pursuing a BSP architecture independent implementation of data structures is worthwhile endeavour that merits future research.

11.2 Less Parallel Slackness

In this report, we presented, for a variety of geometric problems, BSP algorithms with speedups considerably faster (approximately linear) than what can be obtained through the standard virtual processor simulation of fine grained algorithms (that is, for $\frac{n}{p}$ larger than $O(1)$). Indeed, experiments with some of the algorithms presented in this report (see appendix C) show that scalable parallel algorithms designed for $\frac{n}{p}$ larger than $O(1)$ (i.e. $\frac{n}{p} > p$ or $\frac{n}{p} > p \ln n$) seem to have considerable practical relevance.

It remains open if it is possible to make our algorithms fully scalable, that is to develop algorithms that achieve optimality for values of p much closer to n than these, i.e., $p = n^{1-\epsilon}$ for any constant ϵ. A promising step towards that direction has already be taken with the design of an iterative deterministic sorting algorithm [75] that achieves constant factor optimal time in communication for an ample range of values of the parameters n, p, g and L, such that $p = n^{1-\epsilon}$ for any arbitrarily small constant ϵ. The implications of this result are significant and manifold, i.e. considerable reduction in parallel slackness required by algorithms involving sorting, dismissal of the probabilistic feature from algorithms that used the high-probability bucketing algorithm of section B.2.

The basic idea behind the deterministic sorting algorithm is an iterative adaptive partitioning technique. We believe that a similar idea can possibly be used in improving some other geometric algorithms.

11.3 Modifying Existing Expected-Time Algorithms to High-Probability Algorithms

In section 9.1.2 we showed how the Voronoi diagram of n points drawn independently from a uniform distribution function in the unit square can be constructed in $O(g\frac{n}{p})$ expected time. The algorithm used the technique of grid-partitioning previously used in [24], appropriately modified to perform efficiently in a parallel environment. Careful examination of the algorithm revealed that by employing ideas similar to that of section 5.3 we can modify the expected-time complexity results to high-probability ones. Moreover, these results seem to be applicable to several other related proximity problems, i.e. nearest-neighbours, largest empty circle, largest empty rectangle.

11.4 Dynamic Computational Geometry

In many application areas such as robotics, graphics, and air traffic control, it is often imperative to determine geometric properties of dynamic systems [13], i.e. systems of moving objects. As these applications usually occur in real-time environments, the value of a fast (parallel) solution is significant. Current research in the subject has barely scratched the surface, and, therefore, we believe that the design of dynamic computational geometric BSP algorithms is a worthwhile endeavour that deserves to be pursued in the future.

11.5 Computational Geometric Problems

Many well-known computational geometric problems await efficient BSP solutions. In what follows we cite some of the problem areas we believe that should be addressed in the future.[50]

- Sorting, Merging and Bucketing.
- Point location.
- Range searching.
- Intersection of line segments, rectangles, and circles.
- Maxima problems.
- Convex hulls.
- Visibility.
- Separability.
- Query proximity.
- Voronoi diagrams.
- Geometric optimization.
- Triangulation of polygons and point sets.
- Isothetic geometry.
- Arrangements.

We are not aware of any effort toward developing architecture independent parallel algorithms for these problems.

11.5.1 An Algorithmic Library

This work is conducted in conjunction with a major project which is currently underway at Oxford Parallel, that of developing a comprehensive and integrated library of BSP algorithms for computationally intensive problems. The goal is to specify, design and implement a core of BSP algorithms in several scientific areas. On this premise we settled on developing a library of efficient BSP geometric algorithms. The main features of the library will be:

[50] The following list has no pretence of being complete and we expect to be changed throughout the course of this research.

- A sizable collection of data types and algorithms in a form which allows them to be used by non-experts.
- A precise and readable specification for each of the data types and algorithms mentioned above.
- Efficient implementations for each of the data types and algorithms.

We hope that this library will narrow the gap between algorithmic research, implementation and teaching.

11.6 A Case Study

The wide range of applicability of computational geometric algorithms covers in particular some problems arising in very large scale integrated systems. Our future plans include the discussion of such an application in depth and aim in showing how the methods developed in this report can be successfully employed for practical problem solving.

11.7 Conclusion

The purpose of this report has been a unified investigation of a wide range of BSP computational algorithms, and the design of a few techniques that are useful for a better understanding of the communication requirements of parallel machines. The computational model adopted is the Bulk-Synchronous Parallel (BSP) model of computation, which deals explicitly with the notion of communication and synchronization among computational threads. The BSP model was originally proposed by Leslie Valiant [165] and subsequently addressed in [27, 115, 116] as a bridge between software and hardware, i.e. architecture independent software capable of delivering scalable performance for a wide class of applications on a variety of parallel computers [165]. The important distinction between the approach that we considered in this report and the previous approaches is that our approach does not depend upon any particular choice of parallel system family but is universally applicable to any parallel machine (i.e., distributed memory system, shared memory multiprocessor, network of workstations).

A Primitive Operations

The design of a complex algorithm relies heavily on a set of building blocks and on the instruments required to compile the blocks into an algorithm. In this section, we examine some of these basic building blocks (primitive operations) discussed in earlier works [74, 115, 165] and we present algorithms for these operations that are suitable for the BSP model.

A.1 Communication Primitives

The processing units communicate via an interconnection network (a router that can deliver messages (packets) in a point to point mode). The performance of the network g is measured in terms 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). In section 2.3, the local primitive operations *read* and *write* were discussed. In this section we present some additional primitive operations involving interprocessor communication, and analyze their BSP time complexity.

A.1.1 Broadcasting

In a *broadcast* operation [165], a particular processing unit sends one message to all other processors. Clearly, sending one copy of the message to each of the p processing units can be realized in $\log_t p$ supersteps by executing a logical t-ary tree for some appropriate t (the exact value of t is likely to be a function of the number of processing units p participating in the broadcast operation, and the values of g and L; for a detailed discussion see section A.3). In every superstep, t copies of each existing copy of the original message are sent to t distinct processing units. Hence, the time complexity of the broadcast operation, on the BSP model, will be,

$$T = O(\log_t p \max\{tg, L\}). \tag{63}$$

Therefore,

Lemma A.1 *Optimality ($O(\frac{n}{p})$ time) can be achieved if $t = O(\frac{n}{pg\log p}\log\frac{n}{pg\log p})$, $L = O(\frac{n}{p\log_t p})$, and $n = \Omega(ptg\log_t p)$.*

Proof: To secure optimality we should have $tg\log_t p = O(\frac{n}{p})$,

$$\begin{aligned} t &= O(\frac{n}{pg\log_t p}) \\ &= O(\frac{n}{\frac{pg\log p}{\log t}}) \\ &= O(\frac{n\log t}{pg\log p}) \\ &= O(\frac{n}{pg\log p}\log\frac{n}{pg\log p}), \end{aligned} \tag{64}$$

and $\log_t pL = O(\frac{n}{p})$,

$$L = O(\frac{n}{p \log_t p}).$$

Furthermore, it is straightforward to verify that $n = \Omega(ptg \log_t p)$ achieves optimality. ■

Note that, the special case $t = p$ (*linear broadcast*) requires just one superstep.
A broadcast operation is specified by the keyword `broadcast`.

A.1.2 Global Broadcasting

In a *global broadcast* operation, every processing unit (in parallel) sends a different message to each other processing units. The time complexity of the global broadcast algorithm, on the BSP model, will be,

$$T = O(\max\{pg, L\}). \tag{65}$$

It is straightforward to verify the pre-mentioned bound; merely, let each processing unit (in parallel) perform a linear-broadcast.

A global broadcast operation is specified by the keyword `global_broadcast`.

A.2 Computational Primitives

In this section some basic computational primitives are examined, and optimal time complexities are established (given sufficient parallel slackness).

A.2.1 Reduction (Prefix-Sum)

The *reduce (prefix-sum)* operation [29, 31, 108] may be defined as follows.

Definition A.1 *The prefix-sum operation takes some binary associative operator $\oplus$, and an ordered set $(a_0, a_1, ..., a_{n-1})$ of n elements, and returns the value $(a_0 \oplus a_1 \oplus ... \oplus a_{n-1})$.*

It is straightforward to verify that, an optimal $O(n)$ sequential algorithm exists for the prefix-sum problem. In a parallel context the prefix-sum algorithm can be thought of as using a t-ary tree, for some appropriate t (see section A.3), and proceeding in two phases, first moving from the leaves to the root, and then back to the leaves, making some computations at each node [143].

Assume p processing units, with $n > p$. Then, each processing unit can calculate sequentially the prefix-sum of an $\frac{n}{p}$-segment in $O(\frac{n}{p})$ time steps. Using a logical t-ary tree, we can proceed from the leaves to the root and back[51] in $2\log_t p$ supersteps. Therefore, the time complexity of the parallel prefix-sum algorithm, on the BSP model, will be,

[51] The backward execution of the t-ary tree is performed to communicate the final result to all processors.

$$T = O(\frac{n}{p}) + O(\log_t p \max\{tg, L\}). \tag{66}$$

Therefore,

Lemma A.2 *Optimality ($O(\frac{n}{p})$ time) can be achieved if* $t = O(\frac{n}{pg\log p}\log\frac{n}{pg\log p})$, $L = O(\frac{n}{p\log_t p})$, *and* $n = \Omega(ptg\log_t p)$.

Proof: Similar to that of lemma A.1. ■

Note that, the special case $t = p$ (*linear reduction*) requires just two supersteps.

The number of processors that are actively working on the reduce operation decreases geometrically each step (upward execution of t-ary tree) and subsequently increases geometrically each step (downward execution of t-ary tree). However, in some Single Instruction Multiple Data (SIMD) architectures, the unscheduled processors must remain idle. In such cases, it is preferable utilizing all the processors at all steps, and, therefore, decreasing the number of required time steps to half [143].

A reduce operation is indicated with the keyword `reduce`.

A.2.2 Maximum-Minimum Finding

An elementary operation widely used in computational geometry is that of identifying - in a non-ordered set - the *maximal/minimal* element. It is straightforward to verify that, an optimal $O(n)$ sequential algorithm exists. The BSP algorithm to be presented uses a t-ary tree, for some appropriate t (see section A.3). Assume p processing units, with $n > p$. Then, each processing unit can calculate sequentially the maximum of an $\frac{n}{p}$-segment of the input in $O(\frac{n}{p})$ time steps.[52] Next, using a logical t-ary tree, we can proceed from the leaves to the root in $\log_t p$ supersteps. The time required to execute the t-ary tree will be,

$$O(\log_t p \max\{tg, L\}). \tag{67}$$

Hence, the time complexity of the minimum/maximum algorithm, on the BSP model, will be,

$$T = O(\frac{n}{p}) + O(\log_t p \max\{tg, L\}). \tag{68}$$

Therefore,

Lemma A.3 *Optimality ($O(\frac{n}{p})$ time) can be achieved if* $t = O(\frac{n}{pg\log p}\log\frac{n}{pg\log p})$, $L = O(\frac{n}{p\log_t p})$, *and* $n = \Omega(ptg\log_t p)$.

[52] The exact number of steps is $\frac{n}{p} - 1$ when the maximum *or* the minimum is to be found, and $\frac{3n}{2p} - 1$ when *both* the maximum *and* the minimum are to be found (i.e. (a) read a pair of elements, (b) compare them, (c) compare the smaller with the minimum and the larger with the maximum).

Proof: Similar to that of lemma A.1. ■

The maximum and minimum operations are indicated, respectively, with the keywords `max` and `min`.

A.2.3 Parallel All-Prefix (Scan)

A powerful primitive operation is the *parallel all-prefix (scan)* operation [29, 31, 108, 165]. The all-prefix operation may be defined as follows.

Definition A.2 *The all-prefix operation takes some binary associative operator $\oplus$, and an ordered set $(a_0, a_1, ..., a_{n-1})$ of n elements, and returns the ordered set $(a_0, a_0 \oplus a_1, ..., a_0 \oplus a_1 \oplus ... \oplus a_{n-1})$.*

It is straightforward to verify that, an optimal $O(n)$ sequential algorithm exists for the all-prefix problem. In a parallel context the all-prefix algorithm can be thought of as using a t-ary tree, for some appropriate t (see section A.3), and proceeding in two phases, first moving from the leaves to the root, and then back to the leaves, making some computations at each node [74, 143].

Assume p processing units, with $n > p$. Then, each processing unit can calculate sequentially the all-prefix of an $\frac{n}{p}$-segment in $O(\frac{n}{p})$ time steps. Using a logical t-ary tree, we can proceed from the leaves to the root and back in $2\log_t p$ supersteps. Obviously, the communication time required is

$$O(\log_t p \max\{tg, L\}). \tag{69}$$

Therefore, the time complexity of the parallel all-prefix algorithm, on the BSP model, will be,

$$T = O(\frac{n}{p}) + O(\log_t p \max\{tg, L\}). \tag{70}$$

Accordingly,

Lemma A.4 *Optimality ($O(\frac{n}{p})$ time) can be achieved if $t = O(\frac{n}{pg\log p}\log\frac{n}{pg\log p})$, $L = O(\frac{n}{p\log_t p})$, and $n = \Omega(ptg\log_t p)$.*

Proof: Similar to that of lemma A.1. ■

The special case $t = p$ (*linear all-prefix*) requires just two supersteps.

Note that, the number of processors that are actively working on the all-prefix problem decreases geometrically each step (upward execution of t-ary tree) and subsequently increases geometrically each step (downward execution of t-ary tree). However, in some Single Instruction Multiple Data (SIMD) architectures the unscheduled processors must remain idle. In such cases it is preferable to utilize all the processors at all steps; therefore, decreasing the number of required time steps to half [143].

An all-prefix operation is specified by the keyword `all_prefix`.

A.2.4 An Immediate Application of All-Prefix

The previous section analyzed the parallel all-prefix operation. Here, a straightforward application, of the all-prefix operation, in computational geometry is discussed. Consider the *line-of-sight* problem [143]: "given a terrain map in the form of a grid of altitudes and an observation point X on the grid, find which points are visible along a ray originating at the observation point."

It is straightforward to verify that, a point on a ray is visible if and only if no other between it and X has a greater vertical angle. To find if any previous point has a greater vertical angle, the altitude of each point along the ray is converted to a vertical angle (with respect to the point X), and an all-prefix operation is executed using the binary associative operator *maximum*, which returns to each point the maximum previous angle. Finally, to determine whether a point is visible we only need to compare its angle to the result of the all-prefix operation [143]. Therefore, for n points on a ray the time complexity of the line-of-sight algorithm, on the BSP model, will be,

$$T = O(\frac{n}{p}) + O(\log_t p \max\{tg, L\}). \tag{71}$$

Note that, optimality can be achieved if t, L, and n are within the bounds of lemma A.4.

A.3 Optimal t-ary Trees

Most of the BSP algorithms presented thus far have run in $O(\frac{n}{p} + \log_t p \max\{tg, L\})$ time, and all of them employed a t-ary tree for some "appropriate" t. The natural question is: what is the value of t that minimizes the running time of these algorithms? The main result is stated in the following theorem.

Theorem A.1 *Any BSP algorithm that executes a logical t-ary tree, minimizes communication and synchronization costs for $t = \min\{p, \lceil \frac{L}{g} \rceil\}$, for all ratios $\frac{L}{g} > 1$.*

Proof: Let $C_{(t,p)}$ denote the communication and synchronization costs incurred when executing a logical t-ary tree on p processors, $t \in [2, 3, ..., p]$. It is straightforward to verify that,

$$C_{(t,p)} = \log_t p \max\{(t-1)g, L\}. \tag{72}$$

Thus, the objective of this proof is to find the value of t that minimizes $C_{(t,p)}$. We distinguish two cases:

[Case $\frac{L}{g} < p$ **]** Clearly, $L \log_t p$ is a strictly decreasing function in t, for all t such that $t \geq 2$. Moreover, it is not difficult to verify that, $(t-1)g \log_t p$ is a strictly increasing function in t, for all $t \geq 2$, i.e.,

$$\frac{d}{dt}[(t-1)g \log_t p] = (\frac{(t-1)g \ln p}{\ln t})'$$

$$
\begin{aligned}
&= \frac{(t-1)' g \ln p \ln t - (t-1) g \ln p (\ln t)'}{\ln^2 t} \\
&= \frac{g \ln p \ln t - g \ln p + t^{-1} g \ln p}{\ln^2 t} > 0, \text{ for all } t \geq 2.
\end{aligned}
$$

Therefore, for all $t \geq 2$, $C_{(t,p)}$ is minimized by setting,

$$(t-1) g \log_t p = L \log_t p, \tag{73}$$

or, equivalently,

$$t = \frac{L}{g} + 1 \approx \lceil \frac{L}{g} \rceil. \tag{74}$$

The above result completes the case $\frac{L}{g} < p$. It now remains to prove the case $\frac{L}{g} \geq p$.

[**Case** $\frac{L}{g} \geq p$] Since $\frac{L}{g} \geq p$ and $t \leq p$, we obtain that,

$$L \log_t p > (t-1) g \log_t p, \text{ for all } t \geq 2.$$

Moreover, we have shown that, $L \log_t p$ is a strictly decreasing function in t, for all t such that $t \geq 2$. Thus, $C_{(t,p)}$ is minimized when t obtains its maximum possible value. This observation implies that $t = p$. Note that, the restriction $t \leq p$ guarantees that the t-ary tree has a physical interpretation.

In both cases the theorem is proved.[53] ■

[The reader should study the relation between theorem A.1 and table 1. A careful examination reveals that $\Theta(t_{optimal}) = \Theta(\frac{L}{g}) = \Theta(degree)$, which implies that $t_{optimal}$ fully utilizes the network capacity, without, however, having an explicit knowledge of the network architecture!]

Setting $t = \frac{L}{g}$, for all $2 \leq \frac{L}{g} \leq p$, in the algorithms of this section, we obtain the following result,

$$T = O(\frac{n}{p} + L \log_{\frac{L}{g}} p), \tag{75}$$

which minimizes[54] at $p = \frac{n \ln \frac{L}{g}}{L}$. It is, therefore, straightforward to verify that, the best running time that can be achieved for these algorithms is,

[53]Note that in order to simplify exposition, in the remainder of this report we assume that $C_{(t,p)} = \log_t p \max\{tg, L\}$.

[54]Note that,

$$\frac{d}{dp}[\frac{n}{p} + L \log_{\frac{L}{g}} p] = -\frac{n}{p^2} + \frac{L}{p \ln \frac{L}{g}},$$

which is equal to zero at $p = \frac{n \ln \frac{L}{g}}{L}$.

$$T = O(\frac{L}{\ln \frac{L}{g}} + L \log_{\frac{L}{g}} \frac{n \ln \frac{L}{g}}{L}). \tag{76}$$

A.4 A Lower Bound

In the previous section, we proved an $O(\frac{n}{p} + L \log_{\frac{L}{g}} p)$ upper bound for the BSP algorithms presented thus far. The natural question is whether this time bound can be improved. We conclude our discussion of this section with an indication that $\Theta(\frac{n}{p} + L \log_{\frac{L}{g}} p)$ is the best that can be achieved for most of these problems, regardless of the number of processors. In particular, the following theorem states that $\Omega(\frac{n}{p} + L \log_{\frac{L}{g}} p)$ time is required to compute the prefix-sum of n numbers on the BSP model, matching the upper bound of section A.3. In the remainder of this section, assume that $2 \leq \frac{L}{g} \leq p$.

Theorem A.2 *Given some binary associative operator $\oplus$, and an ordered set $S = (a_0, a_1, ..., a_{n-1})$ of n elements, the prefix-sum computation $(a_0 \oplus a_1 \oplus ... \oplus a_{n-1})$ of S on the BSP model requires $\Omega(\frac{n}{p} + L \log_{\frac{L}{g}} p)$ time.*

Proof: Assume an arbitrarily large (fixed) number of processors p. We shall show that the execution time for the prefix-sum computation of n elements is bounded below by $\Omega(\frac{n}{p} + L \log_{\frac{L}{g}} p)$. The term $\Omega(\frac{n}{p})$ follows trivially; in the remainder of this proof we will establish the $\Omega(L \log_{\frac{L}{g}} p)$ bound.

Let the fastest algorithm complete within $s + 1$ supersteps (phases), of time $t_0, t_1, ..., t_s$. In any superstep of time t, each processor can at best compute the prefix-sum of t elements, and, thus, the number of partial prefix-sums reduces by a factor of t at best. Clearly, to compute the prefix-sum of n elements, we must have $\frac{n}{t_0(t_1 ... tn)} \leq 1$, where $t_0 \leq \frac{n}{p}$ is the time required for 0-th superstep - the distinction between t_0 and $t_{i \in [1..s]}$ is essential since the computation in superstep 0 does not require any communication. (In the remainder of this proof, we shall assume that t_0 is as large as $\frac{n}{p}$, without affecting the lower bound, since $\Omega(\frac{n}{p})$ is a lower bound for the prefix-sum computation.) It is not difficult to verify that the computation time $t_{cp} = t_0 + \sum_{i=1}^{s} t_i$ is minimized when $t_i = (\frac{n}{t_0})^{\frac{1}{s}}$ for all $0 < i \leq s$. Therefore, the running time T, to produce the prefix-sum of n elements, is

$$T \geq t_0 + s \max\{(\frac{n}{t_0})^{\frac{1}{s}} g, L\}.$$

[Notice that, if at superstep s_i the number of partial prefix-sums is reduced by a factor of t_i, then at least t_i elements have been communicated at some superstep s_j, such that $0 \leq j < i$, $0 < i \leq s$.]

Let $\beta = s \frac{\log \frac{L}{g}}{\log \frac{n}{t_0}}$. If $\beta \geq 1$, then $s \geq \frac{\log \frac{n}{t_0}}{\log \frac{L}{g}}$, and, therefore,

$$\begin{aligned} T &\geq sL \\ &\geq L \frac{\log \frac{n}{t_0}}{\log \frac{L}{g}} \end{aligned}$$

$$
\begin{aligned}
&\geq L \log_{\frac{L}{g}} \frac{n}{t_0} \\
&\geq L \log_{\frac{L}{g}} p \\
&= \Omega(L \log_{\frac{L}{g}} p). \qquad (77)
\end{aligned}
$$

If $\beta < 1$, then,

$$
\begin{aligned}
T &\geq s(\frac{n}{t_0})^{\frac{1}{s}} g \\
&= g(\beta \log_{\frac{L}{g}} \frac{n}{t_0})(\frac{n}{t_0})^{\frac{\log_{\frac{n}{t_0}} \frac{L}{g}}{\beta}} \\
&= g\beta(\frac{L}{g})^{\frac{1}{\beta}} \log_{\frac{L}{g}} \frac{n}{t_0} \\
&\geq g\frac{L}{1.5g} \log_{\frac{L}{g}} \frac{n}{t_0} \\
&= \Omega(L \log_{\frac{L}{g}} p), \qquad (78)
\end{aligned}
$$

since $\beta x^{\frac{1}{\beta}} > \frac{1}{1.5}x$ for all $\beta < 1$, $x > 1.5$.
[Notice that

$$
\begin{aligned}
\frac{d}{d\beta}[\beta^\beta] &= \beta^\beta(\beta \ln \beta)' \\
&= \beta^\beta(\ln \beta + 1).
\end{aligned}
$$

Therefore, β^β minimizes at $\beta = \frac{1}{e}$, since $\frac{d}{d\beta}[\beta^\beta] = 0$ at $\beta = \frac{1}{e}$ and $\frac{d^2}{d\beta^2}[\beta^\beta] > 0$ for all $0 < \beta < 1$.

Hence, since $\beta < 1$, we have that

$$
\begin{aligned}
\beta x^{\frac{1}{\beta}} &= (\beta^\beta x)^{\frac{1}{\beta}} \\
&\geq (\min_\beta \{\beta^\beta\} x)^{\frac{1}{\beta}} \\
&= ((\frac{1}{e})^{\frac{1}{e}} x)^{\frac{1}{\beta}} \\
&\geq (\frac{1}{1.5}x)^{\frac{1}{\beta}} \\
&\geq \frac{1}{1.5}x,
\end{aligned}
$$

for all $x > 1.5$.]
In either case, the theorem is proved. ■

This lower bound can be applied to any n input, m output computation ϕ, where (a) at least one of the outputs of ϕ depends on all the inputs, and (b) the basic computational step permitted is combining two partial results to get one.

B BSP Sorting, Merging, and Bucketing

The problems of sorting, merging, and bucketing have been studied extensively in the literature because of their many important applications and because of their intrinsic theoretical significance [4, 5, 10, 22, 48, 39, 74, 91, 105, 112, 141]. Many solution strategies have been studied in some depth and their actual performances reported. In this section, we restrict ourselves to the modest goal of providing optimal BSP algorithms for the case $n^{1-\epsilon} = p$ for some $0 < \epsilon < 1$.

B.1 Merging

Merging is a fundamental operation on ordered sets: given two ordered sets A and B, compute the ordered set C resulting from their union[55] $C = A \cup B$ [39]. The problem of merging can be solved by a straightforward sequential algorithm that traverses the two sets in time $O(m + k)$, where $m = |A|$ and $k = |B|$. However, merging is much more difficult to solve efficiently in parallel. Our algorithm follows the same scheme as other efficient merging algorithms. We first describe the general (non-optimal) scheme and then give the basic (optimal) version of the algorithm. Although the first algorithm is correct, it does not achieve optimal running time because the processor workload may be unbalanced. We show how to remedy the problem by attending to certain details. Subsection B.5.3 gives a lower bound for BSP merging, which proves our merging algorithm to be BSP-optimal.

B.1.1 A Simple Merging Algorithm

Let $A = (a_1, a_2, ..., a_m)$ and $B = (b_1, b_2, ..., b_k)$ be two non-decreasing sequences whose elements are drawn from a linearly ordered set S. Without loss of generality, assume that all the elements appearing in the two sorted sequences are distinct. We consider, in this section, the problem of merging these two sequences into a sorted sequence - say, $C = (c_1, c_2, ..., c_{m+k})$. We start by introducing a few definitions. The *rank* of an element a in a given ordered set A, denoted by $r(a : A)$, is the number of elements of A that are less than or equal to a, i.e., $A[r(a : A)] = a$. *Ranking* a sequence $A = (a_1, a_2, ..., a_m)$ in B amounts to computing the integer array $r(A : B) = (r_1, r_2, ..., r_m)$, where $r_i = r(a_i : B)$.

It is not difficult to verify that the merging problem can be reduced to that of determining the rank of each element x from A and B in the ordered set $C = A \cup B$ [96]. Obviously, if $r(x : C) = i$, then $c_i = x$, where c_i is the i-th element of the desired sorted sequence. Moreover, since $r(x : C) = r(x : A) + r(x : B)$, we can merge A and B by calculating the two integer arrays $r(A : B)$ and $r(B : A)$. We now proceed to describe an algorithm to determine $r(A : B)$; by symmetry, the same algorithm can be employed to determine $r(B : A)$.

Essentially, to merge two sequences A and B using p processors, one sequence - say, A - is partitioned into p equal segments, and each processor searches the other data set, in this case B, using a binary search method to determine the rank in B

[55] Note that in the remainder of this section we shall use the $\cup$ (union) operation on ordered sets to denote the merge operation on ordered sets.

of the lower and upper bound elements of its segment in A. Each processor can then apply an optimal-time sequential algorithm to each pair of subsequences to generate the sorted subsequence of an easily computed segment of the destination array. The details of the algorithm follow. Assume that the input arrays A and B are initially distributed among the p processors equally but possibly arbitrarily (uniformly).

Algorithm B.1

Input: *Two arrays (sequences)* $A = (a_1, a_2, ..., a_m)$ *and* $B = (b_1, b_2, ..., b_k)$ *in increasing order.*
Output: *The ordered array (sequence)* C *resulting from their union* $C = A \cup B$.

> **Stage–1** Each processor p_i computes $r(a_{\frac{(i+1)m}{p}} : B)$, by using a binary search procedure.
>
> **Stage–2** Each processor p_i sends $r(a_{\frac{(i+1)m}{p}} : B)$ to processor p_{i+1}, for all $0 \leq i < p-1$.
>
> **Stage–3** Every processor knows which pair of subsequences has to merge and performs this merging (using an optimal-time sequential algorithm). [Notice that processor p_i might be required to communicate (fetch) a data segment of size $r(a_{\frac{(i+1)m}{p}} : B) - r(a_{\frac{im}{p}} : B)$.]

The correctness of the algorithm follows directly from the discussion preceding it.

The time complexity of the algorithm may be analyzed as follows. Let $n = m+k$. The number of iterations executed at stage 1 is $O(\log k)$. The algorithm does not guarantee exclusive read memory accesses, so, the time complexity of stage 1 is at worst $O(\log k \max\{pg, L\})$, and, clearly, dominates that of stage 2. The bottleneck of this algorithm is stage 3 since its time complexity depends on the distribution of A and B; the best-case running time for stage 3 is $O(\max\{g\frac{m+k}{p}, L\})$ (this time bound results when the algorithm partitions each of A and B into equally sized segments), and, the worst-case time complexity is $O(\max\{g(\frac{m}{p}+k), L\})$ (when the algorithm produces a partitioning that is totally unbalanced). Consequently, the worst-case performance of the naive merging algorithm is bounded above by $O(gn + \log k \max\{pg, L\})$, which is a strictly increasing function in p. In theorem B.5.3, it is shown that the lower bound for BSP merging is $\Omega(\log\log n + \max\{g\frac{n}{p}, L\})$, and, therefore, this algorithm is not optimal. In the following section, a BSP-optimal merging algorithm is presented.

B.1.2 Balanced Merging

The balanced parallel merging algorithm avoids the $O(gn)$ bottleneck of the simple merging algorithm by always finding subsequences in A and B which have a total size of $\frac{m+k}{p}$. This is accomplished by adjusting the upper and lower boundary elements of the subsequences in both A and B, rather than fixing those in A, so that each processor is guaranteed to merge a pair of subsequences of summed size $\Theta(\frac{m+k}{p})$. Such pairs of subsequences can then be merged independently to generate the sorted subsequence of an easily computed segment of the destination array. The problem of parallel merging can, therefore, be reduced to that of determining the lower and upper boundaries of these balanced segments (subsequences). The computation of these particular segments involves, however, a sophisticated binary search method. The remainder of this section concentrates on developing such a procedure.

Theorem B.1 *Given two non-decreasing sequences $A = (a_1, a_2, ..., a_m)$ and $B = (b_1, b_2, ..., b_k)$, whose elements are drawn from a linearly ordered set S, it is always possible to determine a set of p segments (possibly empty) $A = (A_1, A_2, ..., A_p)$, and $B = (B_1, B_2, ..., B_p)$ in each subsequence, such that (i) the summed size of corresponding pairs differs by at most one, and that (ii) the elements contained in $A_i \cup B_i$ are smaller than (or equal to) the elements in $A_j \cup B_j$, iff $i < j$, for all $0 \le i, j < p \le k + m$.*

Proof: We start by proving that such a set of segments does exist. Let C be the non-decreasing sequence resulting from the union (merging) of A and B. Clearly, C can always be partitioned in p segments $C_1, C_2, ..., C_p$ such that the size of these segments differs by at most one, and that the elements contained in C_i are smaller than the elements in C_j, where $i < j$ and $0 \le i, j < p$. Therefore, since each C_i can be decomposed into two segments (possibly empty) A_i and B_i, the existence of a p-segment set satisfying conditions (i) and (ii) follows trivially.

It now remains to show how to determine this p-segment set. Consider an arbitrary prefix C' of the non-decreasing sequence C. Let $\gamma = r(c : C)$ denote the rank of the last element c of C' in C, then, from the previous analysis, it follows that there exist two prefixes (possibly empty) A' and B', from A and B respectively, such that $\gamma = r(a : A) + r(b : B)$, and $C' = A' \cup B'$, where $r(a : A) = |A'|$ and $r(b : B) = |B'|$. Let $\alpha = r(a : A)$ and $\beta = r(b : B)$ denote, respectively, the rank of a and b. We are now ready to present the main result of this theorem.

Given some fixed γ we can compute the ranks α and β, such that $\gamma = \alpha + \beta$, and $C' = A' \cup B'$, with the following recursive search rule.

$$\alpha \leftarrow \alpha_{i+1} \leftarrow \begin{cases} \alpha_i < \alpha_{i+1} \le \alpha_{max}, & \text{if } A[\alpha_i] < B[\beta_i] \\ \alpha_{i+1} = \alpha_i, & \text{if } A[\alpha_i] = B[\beta_i] \\ \alpha_{min} \le \alpha_{i+1} < \alpha_i, & \text{if } A[\alpha_i] > B[\beta_i], \end{cases}$$

$$\beta \leftarrow \beta_{i+1} = \gamma - \alpha_{i+1},$$

where α_{min}, α_{max}, α_0, and β_0 are initialized, respectively, to $\max\{1, \gamma - |B|\}$, $\min\{|A|, \gamma\}$, $(\alpha_{min} + \alpha_{max})/2$, and $\gamma - \alpha_0$.

An efficient search procedure can be designed by taking a new value of α_{i+1} to equally partition the remaining search space, i.e., in the first case we could select $\alpha_{i+1} = (\alpha_i + \alpha_{max})/2$. [Notice that, after each iteration of the search procedure the upper and lower bounds of the search space (α_{max} and α_{min} respectively) should be updated in a similar fashion to that of the simple binary search procedure, i.e. in the first case we should set $\alpha_{min} = \alpha_i$.]

The procedure should terminate when, either

$$\alpha_{i+1} = \alpha_i$$

or

$$\alpha_{min} + 1 \geq \alpha_{max}$$

is true. [Notice that the termination criteria are identical to that of simple binary search.]

The correctness of the above search procedure is established next. The case $A[\alpha_i] = B[\beta_i]$ is trivial. The other two cases can be proved by contradiction. Suppose that $A[\alpha_i] < B[\beta_i]$, implies that $\alpha_{min} \leq \alpha < \alpha_i$ (rather than $\alpha_i < \alpha \leq \alpha_{max}$). By definition, $B[\beta] \leq A[\alpha+1]$, and, therefore, we have that $B[\beta] \leq A[\alpha_i] < B[\beta_i]$, which implies that $\beta < \beta_i$. However, this contradicts the assumption that $\alpha_{min} \leq \alpha < \alpha_i$, since by definition $\alpha + \beta = \alpha_i + \beta_i = \gamma$, which implies that $\beta > \beta_i$. The other case follows in a similar fashion.[56]

Hence, choosing $\gamma = (j+1)\frac{n}{p}$, for all $0 \leq j < p$, and selecting the α and β values using the above searching method, guarantees a p-segment partitioning scheme satisfying the conditions of theorem B.1. ■

Corollary B.1 *Given any two non-decreasing sequences of arbitrary size and distribution, A and B, it is always possible to determine a p-segment set in each such that (i) the summed size of the corresponding pairs differs by at most one, and that (ii) the elements are ordered between pairs, with the following recursive search rule,*

$$\alpha \leftarrow \alpha_{i+1} \leftarrow \begin{cases} \alpha_i < \alpha_{i+1} \leq \alpha_{max}, & \textit{if } A[\alpha_i] < B[\beta_i] \\ \alpha_{i+1} = \alpha_i, & \textit{if } A[\alpha_i] = B[\beta_i] \\ \alpha_{min} \leq \alpha_{i+1} < \alpha_i, & \textit{if } A[\alpha_i] > B[\beta_i], \end{cases}$$

$$\beta \leftarrow \beta_{i+1} = \gamma - \alpha_{i+1},$$

where all variables are defined as in theorem B.1.
The procedure should terminate when, either

$$\alpha_{i+1} = \alpha_i$$

or

$$\alpha_{min} + 1 \geq \alpha_{max}$$

is true.

[56] Notice that, if $A[\alpha_i] = B[\beta_i]$ then more than one partitioning schemes might be acceptable.

Proof: Immediate. ■

This search procedure can, therefore, by substituted in the algorithm of section B.1.1 (stage 1) to provide a well behaved (balanced) merging algorithm. The details of the algorithm follow. Assume that the input arrays A and B are initially distributed among the p processors equally but possibly arbitrarily (uniformly).

Algorithm B.2

Input: *Two arrays (sequences)* $A = (a_1, a_2, ..., a_m)$ *and* $B = (b_1, b_2, ..., b_k)$ *in increasing order.*
Output: *The ordered array (sequence)* C *resulting from the union* $C = A \cup B$.

> **Stage–1** Each processor p_i computes the pair $(\alpha_{i+1}, \beta_{i+1})$ by using the search procedure of corollary B.1, where $\gamma_{i+1} = \alpha_{i+1} + \beta_{i+1}) = (i+1)\frac{m+k}{p}$ and $0 \leq i < p$.
>
> **Stage–2** Each processor p_i sends the pair $(\alpha_{i+1}, \beta_{i+1})$ to processor p_{i+1}, for all $0 \leq i < p-1$.
>
> **Stage–3** Every processor knows which pair of subsequences has to merge and performs this merging (using an optimal-time sequential algorithm). [Notice that processor p_i might be required to communicate (fetch) a data segment of size $\alpha_{i+1} + \beta_{i+1} - \alpha_i - \beta_i = \frac{m+k}{p}$.]

The correctness of the algorithm follows directly from the discussion preceding it. The time complexity of the algorithm may be analyzed as follows. Let $n = m + k$, and assume without loss of generality that $k = \min\{k, m\}$. The number of iterations executed in stage 1 is $O(\log k)$, since we can always perform the balanced search procedure on the smaller of the two sequences. The algorithm does not guarantee exclusive read memory accesses, so, the time complexity of stage 1 is at worst $O(\log k \max\{pg, L\})$, and, clearly, dominates that of stage 2. Finally, the running time of stage 3 is $O(\max\{g\frac{n}{p}, L\})$, since, by using the balanced binary search procedure, each processor is guaranteed to merge a pair of subsequences of summed size $\Theta(\frac{m+k}{p})$.

The following theorem summarizes our result.

Theorem B.2 *There exists a BSP algorithm that, given two sorted sequences* A *and* B*, can compute the sorted sequence* $C = A \cup B$ *in*

$$O(\log k \max\{pg, L\} + g\frac{n}{p})$$

time, where $n = m + k$*, and* $k = \min\{m, k\}$.

Proof: Immediate. ■

Corollary B.2 *The merging problem can be solved in BSP-optimal*

$$O(g\frac{n}{p})$$

time, for any integers n*,* p*,* L*, and* g *such that* $p = O(\sqrt{\frac{n}{\log n}})$ *and* $L = O(\frac{gn}{p \log n})$*.*

Proof: In theorem B.5.3, it is shown that the lower bound for BSP merging is $\Omega(\max\{g\frac{n}{p}, L\})$, and, therefore, the above result follows trivially from theorem B.2. ■

Notice that, algorithm B.2 does not guarantee exclusive read memory accesses, so, the time complexity of stage 1 is at worst $O(\log k \max\{pg, L\})$. By employing some of the techniques presented in [74] (for simulating CRCW PRAM on the BSP) and by replicating information, the time complexity of stage 1 may be reduced to $O(\log k \max\{g \log p, L\})$. Nevertheless, the non-trivial details of such an approach seem to be beyond the scope of this report.

A merge operation is specified by the keyword `merge`.

B.2 Bucketing

In this report we shall repeatedly allude to the intimate connection between computational geometry and *bucketing*[57] *(sorting)*. This section affords us the opportunity to present (and analyze) a BSP bucketing algorithm. The optimal high-probability algorithm presented in this section derives from the technique of *over-sampling*, previously utilized in [141] in a parallel context, and the ideas of [72, 74].

The distinctive properties of the buckets utilized in this report are summarized below - assume that buckets are indexed as $(B_0, B_1, ..., B_k)$,

- the elements contained in bucket B_i are smaller than the elements contained in bucket B_j, iff $i < j$ for all $0 \leq i, j \leq k$,
- buckets are of almost equal size with high probability.

Both properties are investigated (in detail) in the remainder of this section.

Let $X = (x_1, x_2, ..., x_n)$ be an ordered set of elements (keys) indexed such that $x_i < x_{i+1}$, for $0 \leq i < n$. The implicit assumption is that keys are unique since we can always make them so, i.e. by appending to them their index position. Let $Y = (y_1, y_2, ..., y_{ks-1})$ be a randomly chosen subset of X also indexed such that $y_i < y_{i+1}$, for $1 \leq i < ks$, for some positive integers k, and s. After determining Y, we may use it to partition the set $X - Y$ into k subsets, $X_0, X_1, ..., X_{k-1}$, where,

$$\begin{aligned} X_0 &= (x : x < y_s), \\ X_i &= (x : y_{is} < x < y_{(i+1)s}), \text{ for all } 1 < i < k-1, \\ X_{k-1} &= (x : y_{(k-1)s} < x). \end{aligned}$$

[57] In most general terms, *buckets* mean the partitions (subregions) into which the entire region under consideration is partitioned [15].

Let n_i be the number of elements in X_i, and $P_i(j)$ be the probability that n_i lies outside j (j being a range), where $1 \le i < k$, and $s-1 \le j \le n-k+1$. In [72] it is proved that,

$$P_i(j) = P(j) = \frac{\binom{j}{s}\binom{n-j}{ks-s}}{\binom{n}{ks}}, \text{ for all } 1 \le i < k. \tag{79}$$

The fact that, $P_i(j) = P(j)$ indicates that $P_i(j)$ is independent of i.

Suppose now that $x < a$, $x < X < A$ are all functions of n such that $xX = o(A)$ and $X = o(A^{\frac{2}{3}})$. Furthermore, let $x = aP \mp G$, and $X = AP \mp G$ where $P = (X+x)/(A+a)$, $G = o(aP)$ and $G = o(AP)$. Then, for all large enough n, it is shown in [141] that,

$$\frac{\binom{a}{x}\binom{A}{X}}{\binom{a+A}{x+X}} \le (1+o(1))e^{-\frac{G^2}{5aP}}, \tag{80}$$

Applying the above facts with $A = n-j$, $a = j$, $X = ks-s$, $x = s$ and letting $s = 7\ln^2 n$, $k = p$ and $j = (1 \pm \frac{1}{\sqrt{\ln n}})(\frac{n}{k})$, we obtain the following,

$$\begin{aligned} (aP) &= (1 \pm \frac{1}{\sqrt{\ln n}})\frac{n}{p}\frac{p7\ln^2 n}{n} \\ &= (1 \pm \frac{1}{\sqrt{\ln n}})7\ln^2 n \\ &= 7\ln^2 n \pm 7\ln^{\frac{3}{2}} n, \end{aligned} \tag{81}$$

and, similarly,

$$\begin{aligned} (AP) &= (n - (1 \pm \frac{1}{\sqrt{\ln n}})\frac{n}{p})\frac{p7\ln^2 n}{n} \\ &= p7\ln^2 n - 7\ln^2 n \pm 7\ln^{\frac{3}{2}} n. \end{aligned} \tag{82}$$

Therefore,

$$G = 7\ln^{\frac{3}{2}} n. \tag{83}$$

Combining now equations 79 and 80, we obtain,

$$\begin{aligned} P(j) &\le (1+o(1))e^{\frac{-G^2}{5aP}} \\ &= (1+o(1))e^{-\frac{(7\ln^{\frac{3}{2}} n)^2}{35\ln^2 n \pm 35\ln^{\frac{3}{2}} n}} \end{aligned}$$

$$= \quad (1+o(1))e^{-1.4\frac{\ln n}{1\pm\frac{1}{\sqrt{\ln n}}}}$$

$$\leq \quad (1+o(1))e^{-\ln n} \tag{84}$$

$$= \quad O(\frac{1}{n}), \tag{85}$$

provided that $p\ln^4 n = o(n)$ and $p\ln^2 n = o(n^{\frac{2}{3}})$; which is true for all $p = n^{\frac{2}{3}-\epsilon}$, for any $0 < \epsilon < \frac{2}{3}$. Note that, inequality 84 holds for all $n \geq e^{6.25} \approx 520$.

We thus have the following lemma.

Lemma B.1 *The probability that the size n_i, $0 \leq i < k$, of any bucket is of size more/less than $(1 \pm \frac{1}{\sqrt{\ln n}})(\frac{n}{k})$ is at most $1 - O(\frac{1}{n})$, provided that $k = n^{\frac{2}{3}-\epsilon}$, for any $0 < \epsilon < \frac{2}{3}$.*

Proof: Immediate. ■

Note that, for different values of s and j we may obtain tighter probability bounds.

The algorithm to be presented for bucketing n items is based on the above lemma and on the sorting algorithm of [74]. The assumption is made that the input is initially distributed among the p processing units equally but possibly arbitrarily (uniformly).

Algorithm B.3

Input: *An array $A_{i=1...n}$ of keys.*
Output: *The "bucketed" array $A_{i=1...n}$.*

Stage–1 Processor p_0 fetches a randomly chosen set of ks elements.

Stage–2 Processor p_0 sorts sequentially the ks elements, and selects $k-1$ splitters $(s, 2s, ..., (k-1)s)$.

Stage–3 Processor p_0 broadcasts the $k-1$ splitters, along with the index of each splitter, so that the processors receive a sorted set of splitters.

Stage–4 Each processor determines the bucket of every element it holds with respect to the $k-1$ splitters using a binary search procedure.

Stage–5 The number of elements that fall into each of the k buckets are counted, and a multiple all-prefix operation is then initiated on the number of elements per bucket and per processor, to find the processor destination of each element.

Stage–6 A global broadcast operation is initiated, where every processor routes the elements in its possession to the pre-determined processor.

The correctness of the above algorithm follows directly from the discussion preceding it. The time complexity of the algorithm can be estimated as follows.

Theorem B.3 *There exists a BSP algorithm that, given a sequence S of size n can compute the bucketed sequence S^* in*

$$\begin{aligned} O(\quad & p \ln^2 n \log (p \ln^2 n) + \\ & \frac{n}{p} \log p + \\ & \max \{gp \ln^2 n, L\} + \\ & \log_t p \max \{tgp, L\} + \\ & \max \{g\kappa \frac{n}{p}, L\}) \end{aligned}$$

time - with probability at least $1 - O(\frac{1}{n})$ *- where* $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: Stage 1 requires $O(\max \{gks, L\})$ time. The sorting in stage 2 can be performed in time $O(ks \log (ks))$, using an optimal sequential sorting algorithm [91, 105]. The broadcast operation in stage 3 requires $O(\log_t p \max \{tgk, L\})$ time. Binary search procedure in stage 4 takes time $O(\frac{n}{p} \log k)$. The multiple all prefix operation (stage 5) can be realized in $O(\frac{n}{p} + \log_t p \max \{tkg, L\})$ time steps (see section A.2.3). Finally, the routing operation in stage 6 takes $O(\max \{g(1 \pm \frac{1}{\sqrt{\ln n}})\frac{n}{k}, L\})$ time. Substituting $t = 2$, $s = 7 \ln^2 n$ and $k = p$ gives the desired result. ■

Note that, algorithm B.3 leaves room for several improvements. For example, the sequential sorting in stage 2 can be improved in a multiplicity of ways (i.e. use recursivelly the parallel algorithm of section B.3). Also, the randomly chosen set need not be constructed in one processor, prior to being distributed for parallel sorting [74].

A bucket operation is specified by the keyword `bucket`.

B.3 High-Probability Sorting

As we recognized in section B.1, it is easy to merge two sorted sequences of summed size n, on a p-processor machine, in $\Theta(g\frac{n}{p})$ steps. Unfortunately, the combination of a $\Theta(g\frac{n}{p})$ merging algorithm with any parallel merge-sort algorithm that requires $O(\log n)$ merge passes, results in $O(g\frac{n \log n}{p})$ time, which is non-optimal (see section B.5.2 for a lower bound on BSP comparison sorting). By allowing however random operations, we will find that the problem of sorting becomes much more tractable. In fact, we will use the randomized bucketing algorithm of section B.2 to devise an algorithm for sorting n elements in optimal $\frac{n \log n}{p}$ time (with high probability) on any BSP machine.

The algorithm of this section is based on the assumption that an optimal $O(n \log n)$ sequential sorting algorithm is available [91, 105]. Additionally, assume that the input is initially distributed among the p processors equally but possibly arbitrarily (uniformly).

We can now proceed with the parallel sorting algorithm.

Algorithm B.4

Input: *A sequence S of size n.*
Output: *The sequence S in sorted order.*

Stage–1 The bucketing technique of section B.2 is used to route the elements of every processor to the appropriate bucket (processor). [Notice that the resulting buckets are of size $< (1+\frac{1}{\sqrt{\ln n}})\frac{n}{p}$, with probability at least $1 - O(\frac{1}{n})$ (see lemma B.1).]

Stage–2 Every processor (in parallel) sorts the elements in its possession, by employing any optimal-time sequential algorithm, i.e. [91, 105, 172].

The correctness of the above algorithm follows directly from the first property of buckets as defined in section B.2.

The time complexity of stage 1 was established in the previous section. It is straightforward to verify that the time complexity of Stage 2 is,

$$(1+\frac{1}{\sqrt{\ln n}})\frac{n}{p}\log((1+\frac{1}{\sqrt{\ln n}})\frac{n}{p}), \tag{86}$$

with probability at least $1 - O(\frac{1}{n})$.[58] Therefore, the overall time complexity of algorithm B.4 can be estimated as follows.

Theorem B.4 *There exists a BSP algorithm that, given a sequence S of size n can compute the sequence S in sorted order in*

$$\begin{aligned} O(\quad & p\ln^2 n \log(p\ln^2 n) + \\ & \frac{n}{p}\log p + \\ & \kappa\frac{n}{p}\log(\kappa(\frac{n}{p})) + \\ & \max\{gp\ln^2 n, L\} + \\ & \log_t p \max\{tgp, L\} + \\ & \max\{g\kappa\frac{n}{p}, L\}) \end{aligned}$$

time - with probability at least $1 - O(\frac{1}{n})$ - where $\kappa = 1 + \frac{1}{\sqrt{\ln n}}$.

Proof: Immediate. ■

Therefore,

[58] Note that, the probability that any bucket is of size more than $(1+\frac{1}{\sqrt{\ln n}})(\frac{n}{p})$ is at most $1 - O(\frac{1}{n})$ (see lemma B.1).

Corollary B.3 *There exists a BSP sorting algorithm that for any integers* n, p, L, *and* g *such that* $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n \log n}{p \log p})$, *and* $g = O(\log n)$ *has time complexity*

$$O(\frac{n \log n}{p}),$$

with probability at least $1 - O(\frac{1}{n})$.

Proof: Let in theorem B.4 $t = 2$. ■

Notice that the algorithm is asymptotically optimal since the lower bound on sequential sorting is $\Omega(n \log n)$. Obviously, by substituting different values for s, j in the analysis of section B.2, a variety of bounds on the runtime of algorithm B.4 can be obtained.

A sort operation is specified by the keyword `sort.`

[To demonstrate the practical relevance of our scalable BSP algorithms, we implemented the above high-probability sorting algorithm on a Silicon Graphics Power Challenge shared memory system (consisting of four 64-bit Streaming Superscalar TFP processors) and an IBM 9076 close-coupled distributed memory system (consisting of eight IBM RS/6000 model 580 processors). The results of our experiments can be found in appendix C section C.2.]

B.4 Deterministic Parallel Sorting

In the past few sections, we have focused on high-probability algorithms for sorting (bucketing) on BSP machines. We now shift our attention to deterministic algorithms for sorting. As we recognized in section B.3, it is hard to sort n items (deterministically), on a p-processor machine, in $\Theta(\frac{n \log n}{p})$ time, since the combination of any $\Theta(g\frac{n}{p})$ merging algorithm with any parallel merge-sort algorithm that requires $O(\log n)$ merge passes, results in $O(g\frac{n \log n}{p})$ time, which is non-optimal.

In this section, we consider a merge-based sorting BSP algorithm that achieves linear speedup, using a bucketing partitioning scheme which splits an arbitrary number of sorted sequences among all the processors almost equally (independently of the input distribution). The partitioning scheme for each sorted sequence is found efficiently (using merging and parallel prefix on a small subset of the input) and only one merge pass is required.

B.4.1 The Algorithm

Let $S = (v_1, v_2, ..., v_n)$ be a sequence of n data elements that is initially distributed among the p processors evenly but possibly arbitrarily. The algorithm consists of the following distinct phases:

- *Local Sorting.* In this phase, each processor (in parallel) sorts the subsequence of size $\frac{n}{p}$ that resides in its local memory, by employing an optimal-time sequential algorithm [91, 105].

- *Partitioning (Bucketing).* In this phase, processors cooperate to evenly split the sorted sequences among the processors (independently of input distribution).
- *Merging.* Each processor (in parallel) merges a small number of subsequences (of summed size $\frac{n}{p}$), by employing an optimal sequential algorithm.

Certain results are required before proceeding to the details of the algorithm.

Lemma B.2 *The problem of merging k sorted subsequences $S_1, S_2, ..., S_k$ into one sorted sequence S can be solved (sequentially) in $O(n \log k)$ time, where n is the total number of elements in all the input subsequences.*

Proof: Let H denote the heap (priority queue) constructed by considering the first element of each S_i, for all $1 < i \leq k$. Then, the sorted sequence S can be obtained from H with the following simple procedure, (a) extract from H the minimal element h_{min} and append it to S, (b) assume that $h_{min} \in S_i$; pop the top (minimum) element of S_i and insert it in H (if S_i is empty then skip this step), (c) repeat steps (a)-(b) until $|S| = n$. The correctness of the this procedure is easy to see.

The time complexity of the above process may be analyzed as follows. The preprocessing step (construction of heap) takes time $O(k \log k)$ [105]. Each iteration of steps (a)-(b) requires $O(\log k)$ time, and since the process takes $n - k$ steps (iterations) the overall cost of steps (a)-(c) is bounded above by $O((n-k)\log k)$. Thus, the problem of merging k sorted subsequences into one sorted sequence can be solved (sequentially) in

$$O((n-k)\log k + k \log k) = O(n \log k) \tag{87}$$

time. We thus have the above lemma. ■

Lemma B.3 *The problem of merging p sorted subsequences $S_1, S_2, ..., S_p$ of size $n \geq p$ (each) into one sorted sequence S can be solved (in parallel) in $O(gp \log np + \log p \log nL + gn \log p)$ time.*

Proof: We use the algorithm of section B.1.2 to merge recursively the p subsequences. Consequently, the worst-case time T to merge p sorted subsequences, each of size n, into one sorted sequence of size pn is (see theorem B.2),

$$\begin{aligned} T &= O(\sum_{i=0}^{\log p - 1} (\log(2^i n) \max\{2^i g, L\} + g\frac{2^i n}{2^i})) \\ &\leq O(\sum_{i=0}^{\log p - 1} g \log(2^i n) 2^i + L \sum_{i=0}^{\log p - 1} \log 2^i n + gn \log p) \\ &= O(gp \log(np) + L \log p \log n + gn \log p). \end{aligned} \tag{88}$$

The lemma follows. ■

We are now ready to present the details of the algorithm.

Algorithm B.5

Input: *A sequence S of size n.*
Output: *The sequence S in sorted order.*

Stage–1 Each processor p_i (in parallel) sorts the subsequence S_i of size $h = \frac{n}{p}$ that resides in its local memory, for all $0 \leq i < p$, by employing any optimal-time sequential algorithm, i.e. [91, 105, 172].

Stage–2 Each processor p_i selects $\omega p - 1$ evenly distributed splitter keys $(s_1^{(i)}, s_2^{(i)}, ..., s_{\omega p-1}^{(i)})$ that partition S_i into ωp segments of equal size.

Stage–3 The p sorted subsequences of $\omega p - 1$ splitters (each) are merged using the parallel merge algorithm of section B.1.2. Let the sequence $M = (s_1, s_2, ..., s_{i\omega p}, ..., s_{\omega p^2 - p})$ be the result of the merge operation. [Notice that this sequence is assumed to be evenly distributed among the processors, i.e. the subsequence $s_{i(\omega p-1)+1} ... s_{(i+1)(\omega p-1)}$ resides on processor p_i, where $0 \leq i < p$.]

Stage–4 A global broadcast operation is initiated, where processor p_i sends splitter $s_{(i+1)(\omega p-1)}$ to all other processors, for all $0 \leq i < p-1$.

Stage–5 Every processor counts the number of elements that fall into each of the p buckets (determined by the $p-1$ splitters that were received in previous stage), and a multiple all-prefix operation is then initiated on the number of elements per bucket and per processor, to find the processor destination of each element.

Stage–6 Every processor knows where to route every element in its possession, and performs this routing.

Stage–7 Each processor (in parallel) merges the p sorted sequences (that were received in the previous stage), using the procedure described in lemma B.2.

The correctness of the algorithm is easy to see. It now remains to prove that stages 3-6 partition S into evenly sized segments (buckets). The main result is summarized in the following lemma.

Lemma B.4 *Algorithm B.5 (stages 3-6) partitions an arbitrary size and distribution input sequence S into p segments of size $(1 \pm \frac{1}{\omega})\frac{n}{p}$, for all $n \geq p^2\omega$, $\omega \geq 1$.*

Proof: Consider an arbitrary splitter $s_{i\omega p}$, $1 \leq i < p$, from the sequence $M = (s_1, s_2, ..., s_{i\omega p}, ..., s_{\omega p^2-p})$ (recall that M is the result of the merge in stage 4). It is easy to verify that, there are at least $\frac{i\omega pn}{\omega p^2} = \frac{in}{p}$ data elements which are not larger than $s_{i\omega p}$ since there are $i\omega p$ segments of size $\frac{n}{\omega p^2}$ (each) whose elements are not larger than $s_{i\omega p}$. Likewise, there are at least $\frac{n(\omega p^2 - i\omega p - p)}{\omega p^2} = n - \frac{in}{p} - \frac{n}{\omega p}$ data elements which are not smaller than $s_{i\omega p}$ since there are $\omega p^2 - i\omega p - p$ segments of size $\frac{n}{\omega p^2}$ (each) whose elements are not smaller than $s_{i\omega p}$.

Thus, the number of elements n_i that are smaller than $s_{i\omega p}$ is bounded by,

$$\frac{in}{p} \leq n_i \leq \frac{in}{p} + \frac{n}{\omega p}. \tag{89}$$

In a similar fashion, we obtain that,

$$\frac{(i+1)n}{p} \leq n_{i+1} \leq \frac{(i+1)n}{p} + \frac{n}{\omega p}. \tag{90}$$

Hence, it is straightforward to verify, that algorithm B.5 creates partitions of size,

$$(1 \pm \frac{1}{\omega})(\frac{n}{p}). \tag{91}$$

The lemma follows trivially. ■

The following theorem summarizes our result.

Theorem B.5 *There exists a BSP algorithm that, given a sequence S of size n can compute the sorted sequence S in*

$$\begin{aligned} O(\quad & gp \log(\omega p^2) + \\ & L \log p \log(\omega p) + \\ & g\omega p \log p + \\ & \log_t p \max\{tgp, L\} + \\ & \frac{n}{p} \log \frac{n}{p} + \\ & \max\{g(1 + \frac{1}{\omega})\frac{n}{p}, L\} + \\ & ((1 + \frac{1}{\omega})\frac{n}{p}) \log p) \end{aligned}$$

time, for any $\omega \geq 1$.

Proof: The sequential solution of the sorting problem in stage 1 requires $O(\frac{n}{p} \log \frac{n}{p})$ time. Stage 3 takes $O(gp \log(\omega p^2) + L \log p \log(\omega p) + g\omega p \log p)$ time (see lemma B.3) and clearly dominates stage 2. In stage 4 we only need to be able to route a p-relation which requires time $O(\max\{gp, L\})$. The multiple all-prefix operation in stage 5 can be realized in $O(\frac{n}{p} + \log_t p \max\{tgp, L\})$ steps. In stage 6 we need to route an $O((1+\frac{1}{\omega})\frac{n}{p})$-relation which requires time $O(\max\{g(1+\frac{1}{\omega})\frac{n}{p}, L\})$. Finally, stage 8 can be completed in $O((1+\frac{1}{\omega})\frac{n}{p}\log p)$ steps (see lemma B.2). We thus obtain the above theorem. ■

Accordingly,

Corollary B.4 *The problem of sorting n data elements can be solved in optimal*

$$O(\frac{n \log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $n = \Omega(p^2 \log p)$, $L = O(\frac{n \log n}{p \log^2 p})$, $g = O(\log n)$.

Proof: Let in theorem B.5 $t = 2$ and choose ω to exceed 1 by a constant. ■

Obviously, by substituting different values for ω in the above proposition, a variety of bounds on the runtime of algorithm B.5 can be obtained. Moreover, in [75], an iterative version of the algorithm is presented that achieves comparable performance for values of p much closer to n than those of corollary B.4, i.e. $p = n^{1-\epsilon}$ for any positive constant $0 < \epsilon < 1$.

A sort operation is specified by the keyword `sort`.

[To demonstrate the practical relevance of our scalable BSP algorithms, we implemented the above deterministic sorting algorithm on a Silicon Graphics Power Challenge shared memory system (consisting of four 64-bit Streaming Superscalar TFP processors) and an IBM 9076 close-coupled distributed memory system (consisting of eight IBM RS/6000 model 580 processors). The results of our experiments can be found in appendix C section C.1.]

B.5 Lower Bounds for BSP Comparison Problems

In this section we will define and analyze a variant of the *parallel binary-comparison* model [164] which we will call the *BSP binary-comparison* model. In the spirit of [164], we only use comparisons between data elements to obtain order information about an input sequence $S = (v_1, v_2, ..., v_n)$. That is, given two elements v_i and v_j, we perform a binary comparison; the output of the comparison is one of the following: $<, =, >$.[59] We may not inspect the values of the elements or obtain order information about them in any other way, i.e., no element can be viewed as a string of bits, and no arithmetic operations on the data elements are allowed. Hence, a typical step of the BSP binary-comparison model involves a set of comparisons made by some or all the available processors, followed by a set of data movements. The major difference between the BSP binary-comparison model and the model

[59]Notice that we may dismiss the equal ($=$) binary comparison by assuming that all input elements are distinct. This is possible, since we can always differentiate between two equal input elements, i.e., by appending to them their index position. Therefore, in the remainder of this section, we assume that input elements are distinct.

described in [164] lies in the way that non-local memory accesses are treated. The details of the BSP binary-comparison model are presented in the following section.

B.5.1 The BSP Decision-Tree Model

Comparison problems can be viewed abstractly in terms of decision trees [46]. A decision-tree reflects the comparisons performed by a sorting (merging) algorithm when it operates on a given size input. Note that this definition suppresses all computational steps that do not involve binary comparisons of input elements. In a decision tree, each internal node represents a binary comparison between two elements v_i and v_j for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence. Each leaf represents a permutation $(v_{\sigma(1)}, v_{\sigma(2)}, ..., v_{\sigma(n)})$. The execution of the sorting (merging) algorithm correspond to tracing a path from the root of the decision tree to one of its leafs, where at each internal node, a binary comparison is performed. The left subtree then prescribes subsequent comparisons for - say, $v_i < v_j$ - and the right subtree prescribes subsequent comparisons for - say $v_i > v_j$. When we reach a leaf, the algorithm has established the arrangement $(v_{\sigma(1)} < v_{\sigma(2)} < ... < v_{\sigma(n)})$.

The *BSP decision-tree model* is an extended decision-tree model which allows algorithms to work as follows. The algorithm consists of rounds called supersteps. In each round binary comparisons are performed simultaneously by all (or a subset of) the p processors. Each internal node v represents a set of p comparisons performed between p pairs of elements (not necessarily distinct), and each of the branches out of v corresponds to one of the 2^p possible outcomes of the comparisons. The input of each comparison consists of two elements. The output of each comparison is one of the following two: $<$, $>$. Comparisons and data movements that only involve elements stored in the local memories of processors will be charged $O(1)$ time and will be designated as *local comparisons (data movements)*. Comparisons and data movements that involve elements not residing in the local memories of processors will be charged $O(g)$ time steps and will be designated as *non-local comparisons (data movements)*. Additionally, each round (superstep) will be charged $O(L)$ time steps (for barrier synchronization).

B.5.2 Lower Bounds for Sorting

The problem of sorting an input sequence of length n was presented in sections B.3 and B.4. In this section, the time complexity of sorting n elements using $p \leq n$ processors on the BSP decision-tree model is considered, and a lower bound is obtained along the lines of [5].

Theorem B.6 *In the BSP decision-tree model, any algorithm that uses p processors to sort n data elements in k rounds (supersteps), by storing only a constant number of copies per data element (i.e. $O(\frac{n}{p})$ storage per processor), requires time,*

$$\Omega(\frac{n \log n}{p} + g(\frac{n \log n}{p \log \frac{n}{p}} - \frac{n}{p}) + kL).$$

Proof: The $\Omega(\frac{n \log n}{p} + kL)$ lower bound on the number of computation and synchronization steps is trivial. To obtain a bound on the communication steps, consider a BSP decision-tree that sorts n data elements. It is straightforward to verify that, since there are $n!$ permutations of the n data elements, each permutation representing a relative ordering (distinct sorting sequence), the tree must have at least $n!$ leaves.

Let $n_i = c_i \frac{n}{p}$ denote the maximum number of data elements residing in the local memory of processor p_i, for some constant $c_i > 0$, and let π_0 denote the number of relative orderings that can be distinguished, at the beginning of the computation (i.e., at superstep 0, when no data elements have been communicated), by only comparing the elements that reside in the local memories of the p_i's, for all $0 \leq p_i < p$. Then,

$$\begin{aligned} \pi_0 &= \prod_{i=0}^{p-1} n_i! \\ &\leq \prod_{i=0}^{p-1} n_i^{n_i} \\ &\leq \prod_{i=0}^{p-1} (c_i \frac{n}{p})^{c_i \frac{n}{p}} \\ &\leq \prod_{i=0}^{p-1} (c \frac{n}{p})^{c \frac{n}{p}}, \end{aligned} \tag{92}$$

where $c = \max_i \{c_i\}$. Furthermore, let m denote the maximum number of communication steps during which processors exchange data elements between them. Then, the additional number of relative orderings π_m that can be distinguished is,

$$\begin{aligned} \pi_m &\leq \prod_{i=0}^{p-1} n_i^m \\ &\leq \prod_{i=0}^{p-1} (c_i \frac{n}{p})^m \\ &\leq \prod_{i=0}^{p-1} (c \frac{n}{p})^m, \end{aligned} \tag{93}$$

where $c = \max_i \{c_i\}$. Therefore, the total number of permutations π that can be distinguished by only comparing the elements that reside in the local memories of the p_i's, for all $0 \leq p_i < p$, is bounded by,

$$\pi = \pi_0 \pi_m$$

$$\begin{aligned}
&\leq \prod_{i=0}^{p-1} (c\frac{n}{p})^{c\frac{n}{p}} \prod_{i=0}^{p-1} (c\frac{n}{p})^{m} \\
&= (c\frac{n}{p})^{pc\frac{n}{p}} (c\frac{n}{p})^{pm} \\
&= (c\frac{n}{p})^{p(c\frac{n}{p}+m)}. \qquad (94)
\end{aligned}$$

Let u denote the total number of relative orderings (i.e., number of distinct permutations) to be distinguished by the BSP sorting algorithm. Then, obviously,

$$\begin{aligned}
u &\leq (c\frac{n}{p})^{p(c\frac{n}{p}+m)} \\
m &\geq \frac{\log u}{p \log (c\frac{n}{p})} - c\frac{n}{p}, \qquad (95)
\end{aligned}$$

and since $u \geq n!$, we obtain,

$$m \geq \frac{n \log n}{p \log (c\frac{n}{p})} - c\frac{n}{p} = \Omega(\frac{n \log n}{p \log \frac{n}{p}} - \frac{n}{p}). \qquad (96)$$

Thus, in the BSP decision-tree model, any algorithm that uses p processors to sort n data elements requires,

$$\Omega(g(\frac{n \log n}{p \log \frac{n}{p}} - \frac{n}{p})), \qquad (97)$$

communication time. The theorem follows. ■

B.5.3 Lower Bounds for Merging

The problem of merging two sorted subsequences of summed size n is considered next. The main result is presented next.

Theorem B.7 *In the BSP decision-tree model, any algorithm that uses p processors to merge two subsequences of summed size n in k rounds (supersteps) requires time,*

$$\Omega(\log \log n + \frac{gn}{p} + kL).$$

Proof: The $\Omega(\log \log n)$ lower bound is shown in [96]. Moreover, it is easy to verify that, there always exists a distribution (permutation) of the input into processors, such that, at least one processor involves $\frac{n}{p}$ non-local data movements. The cost of these non-local data operations is bounded below by,

$$\Omega(g\frac{n}{p}). \qquad (98)$$

Thus, in the BSP decision-tree model, any algorithm that merges two subsequences of summed size n in k rounds (supersteps), employing p processors, requires time,

$$T = \Omega(\log \log n + \frac{gn}{p} + kL).$$

The result follows. ■

C Experimental Results

Throughout this section we will use a set of sample problems compiled from various sources to compare the different algorithms and to examine their behaviour with respect to varying problem sizes and processor nodes. As noted in [144], "to choose a suitable test one has to decide between contradicting goals:"

- Too many computational results might tire the reader.
- Too few results might not exhibit enough insight.
- Not all test instances are suitable for every problem.
- Problems from different sources might not be comparable in a fair way.

To overcome these problems we adopted the following framework [144].

- We have selected a set of test problem instances which can always be handled by our algorithms. This set consists of 18 Euclidean problem instances in one-, two-, and three-dimensional space of sizes ranging from 400 to 1,024,000 nodes (vectors) and is given in table 3.
- We have tried to dispense of random problem instances. We have only included random problems[60] for two reasons: (a) for testing algorithms that were designed under the assumption of a certain input distribution (b) for extrapolating CPU times. To our opinion, this should be the only reasons for favouring random test instances. Obviously, the treatment of real problem instances should be of primary concern, because these are the problems whose solution is of interest. Moreover, real problem instances, have qualities that cannot be duplicated by random test samples.

Concerning the CPU times that are either explicitly given or presented in a graphical exhibit, the following remarks apply.

- All CPU times are given in seconds on (a) a Silicon Graphics Power Challenge shared memory system, consisting of four 64-bit Streaming Superscalar TFP processors, for a total capacity of 1.2Gflops peak with

[60] For these problem instances, the random points are located in a square and are drawn independently from a uniform distribution.

Problem Instances			
Problem	Size	Dimension	Source
`1Ds32`	32,128	1D	Statistical Database
`1Ds64`	63,145	1D	Statistical Database
`1Dg128`	129,123	1D	Geographical Database
`1Dg256`	257,871	1D	Geographical Database
`1Dg512`	514,281	1D	Geographical Database
`1Dg1024`	1,040,903	1D	Geographical Database
`2Ds16`	15,871	2D	Statistical Database
`2Ds32`	34,410	2D	Statistical Database
`2Ds64`	63,290	2D	Statistical Database
`2Dg128`	127,981	2D	Geographical Database
`2Dg256`	256,630	2D	Geographical Database
`2Dg512`	513,755	2D	Geographical Database
`3Dc0.4`	402	3D	CAD Database
`3Dc0.8`	873	3D	CAD Database
`3Dc1.6`	1,654	3D	CAD Database
`3Dm3.2`	3,128	3D	3D-Modelling Database
`3Dc6.4`	6,431	3D	CAD Database
`3Dc12.8`	12,239	3D	CAD Database

Table 3: Sample Problems.

512MBytes RAM, (b) an IBM 9076 close-coupled distributed memory system, consisting of eight IBM RS/6000 model 580 processors, each with 128MByte RAM, connected by a High Performance Switch, for a total capacity of 1Gflops peak.

· All software (unless otherwise indicated) has been written in C - by the author - and was linked with the Oxford BSP Library Version 1.2 (see [125]). The source code was compiled using `cc` with option `-O3` under the operating systems `IRIX 6.0.1` and `AIX 3.2` on the Silicon Graphics and IBM systems respectively. The function `bsp_time` was used for measuring running times (see [125] for details on `bsp_time`).

· Distance matrices and lookup tables are not used. Distances, angles, and polar coordinates are always computed by evaluating Euclidean functions and not by matrix or table lookup.

For each test problem instance, we performed a series of 8 experiments, and the running times indicated are the average of those 8 experiments.

Throughout this section the following notation will be used.

· [n] global problem size.

· [p] number of processing units.

· [t_i] running time of parallel algorithm on i processing units, where $2 \leq i \leq p$.

Silicon Graphics Power Challenge				
Problem	n	t_{seq}	t_2	t_4
`1Ds32`	32K	0.27	0.16	0.09
`1Ds64`	64K	0.58	0.34	0.17
`1Dg128`	128K	1.25	0.71	0.36
`1Dg256`	256K	2.72	1.48	0.78
`1Dg512`	512K	5.79	2.97	1.50
`1Dg1024`	1,024K	12.10	6.21	3.12

Table 4: Timing results for algorithm B.5 [deterministic sorting] on a 4 node Silicon Graphics Power Challenge shared memory system.

- [t_{seq}] running time of sequential algorithm (note that, the term t_{seq} appearing in this definition is not necessarily the value of t_i for $i = 1$).

C.1 BSP Deterministic Sorting

In section B.4 we considered a merge-based BSP sorting algorithm that achieves linear speedup, using a bucketing partitioning scheme which splits an arbitrary number of sorted sequences among the processors almost equally, independently of the input distribution. The partitioning scheme for each sorted sequence is found efficiently (using merging and parallel prefix on a small subset of the input) and only one merge pass is required. Moreover, it was proved that the algorithm requires

$$O(\frac{n \log n}{p})$$

time, on the BSP model, for any integers n, p, L, and g such that $n = \Omega(p^2 \log p)$, $L = O(\frac{n \log n}{p \log^2 p})$, $g = O(\log n)$. In this section we present a BSP implementation (in C) of the algorithm, and analyze experimentally its practical performance.

C.1.1 Performance

The theoretical asymptotic performance of the deterministic sorting algorithm is fundamentally $O(\frac{n \log n}{p})$ for appropriately bounded n, p, g, and L. We tested our algorithm on two different machines with a varying number of nodes and input sizes. The results are summarized in tables 4 and 5.[61]

Figures 18 and 19 compare the optimal and actual speedup of algorithm B.5 [deterministic sorting] for sorting 32,000 to 1,024,000 elements on a Silicon Graphics Power Challenge shared memory system and an IBM 9076 SP1 close-coupled distributed memory system respectively. The nearly linear speedup is evident in both figures.

The relative efficiencies of the 4 node SG Power Challenge implementation and the 8 node IBM 9076 SP1 implementation are compared (graphically) in figure 20. The nearly optimal efficiency is evident in both cases.

[61] Sequential sorting is performed using the standard C library routine `qsort`.

IBM 9076 SP1					
Problem	n	t_{seq}	t_2	t_4	t_8
`1Ds32`	32K	0.32	0.18	0.09	0.06
`1Ds64`	64K	0.56	0.34	0.17	0.09
`1Dg128`	128K	1.27	0.74	0.39	0.21
`1Dg256`	256K	2.72	1.43	0.74	0.38
`1Dg512`	512K	6.02	3.18	1.57	0.78
`1Dg1024`	1,024K	11.98	6.12	3.05	1.54

Table 5: Timing results for algorithm B.5 [deterministic sorting] on an 8 node IBM 9076 SP1 close-coupled distributed memory system.

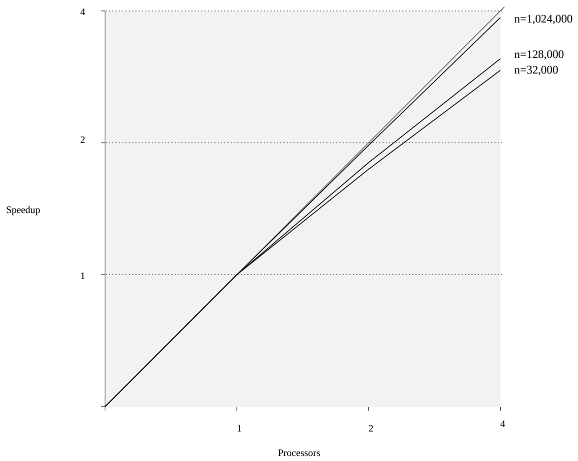

Figure 18: Speedup achieved by algorithm B.5 [deterministic sorting] on a 4 node Silicon Graphics Power Challenge shared memory system.

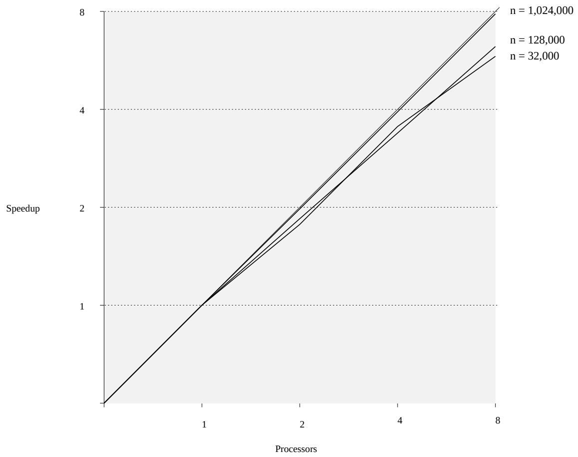

Figure 19: Speedup achieved by algorithm B.5 [deterministic sorting] on an 8 node IBM 9076 SP1 close-coupled distributed memory system.

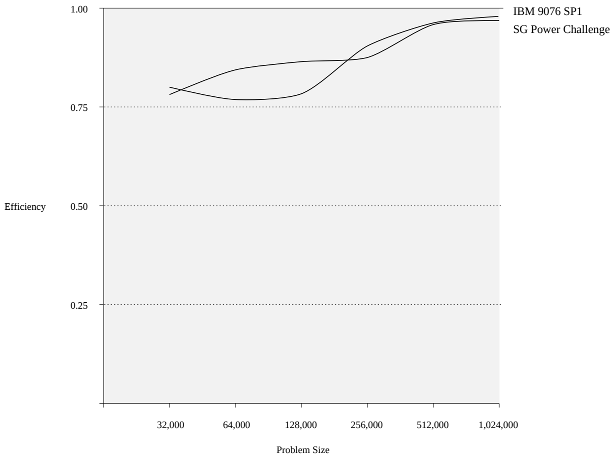

Figure 20: Efficiency of algorithm B.5 [deterministic sorting] on a 4 node Silicon Graphics Power Challenge shared memory system and an 8 node IBM 9076 SP1 close-coupled distributed memory system.

Silicon Graphics Power Challenge				
Problem	n	t_{seq}	t_2	t_4
`1Ds32`	32K	0.27	0.15	0.10
`1Ds64`	64K	0.58	0.31	0.18
`1Dg128`	128K	1.25	0.65	0.34
`1Dg256`	256K	2.72	1.41	0.71
`1Dg512`	512K	5.79	2.98	1.49
`1Dg1024`	1,024K	12.10	6.12	3.07

Table 6: Timing results for algorithm B.4 [high-probability sorting] on a 4 node Silicon Graphics Power Challenge shared memory system.

C.2 BSP High-Probability Sorting

In section B.3 we had the opportunity to present and analyze a high-probability BSP sorting algorithm. By allowing random operations, we concluded that the problem of sorting becomes much more tractable. In particular, we used the randomized bucketing algorithm of section B.2 to devise an algorithm for sorting n elements in optimal

$$O(\frac{n \log n}{p})$$

time with probability at least $1 - O(\frac{1}{n})$ - for any integers n, p, L, and g such that $p = O(\frac{\sqrt{n}}{\ln n})$, $L = O(\frac{n \log n}{p \log p})$, and $g = O(\log n)$. The basic idea behind the algorithm derives from the technique of *over-sampling*, previously utilized in [141] in a parallel context, and the ideas of [72, 74]. In this section we present a BSP implementation (in C) of these algorithmic ideas, and analyze experimentally their practical significance.

C.2.1 Performance

The theoretical asymptotic performance of the high-probability sorting algorithm is fundamentally $O(\frac{n \log n}{p})$ (with high-probability) for an ample range of values of n, p, g, and L. To determine the practical significance of this complexity bound, we tested our algorithm on two different machines with a varying number of nodes and input sizes. The results for 6 sample problems are presented in tables 6 and 7.[62]

Figures 21 and 22 present the optimal and actual speedup of algorithm B.4 [high-probability sorting] for 6 sample problems of size ranging from 32,000 to 1,024,000 elements on a Silicon Graphics Power Challenge shared memory system and an IBM 9076 SP1 close-coupled distributed memory system respectively. The nearly linear speedup is evident in both figures.

The relative efficiencies of the 4 node SG Power Challenge implementation and the 8 node IBM 9076 SP1 implementation are compared (graphically) in figure 23. The nearly optimal efficiency is evident in both cases.

[62] Sequential sorting is performed using the standard C library routine `qsort`.

IBM 9076 SP1					
Problem	n	t_{seq}	t_2	t_4	t_8
`1Ds32`	32K	0.32	0.17	0.09	0.05
`1Ds64`	64K	0.58	0.33	0.17	0.09
`1Dg128`	128K	1.27	0.73	0.37	0.19
`1Dg256`	256K	2.72	1.40	0.71	0.36
`1Dg512`	512K	6.02	3.10	1.53	0.77
`1Dg1024`	1,024K	11.98	6.08	3.05	1.53

Table 7: Timing results for algorithm B.4 [high-probability sorting] on an 8 node IBM 9076 SP1 close-coupled distributed memory system.

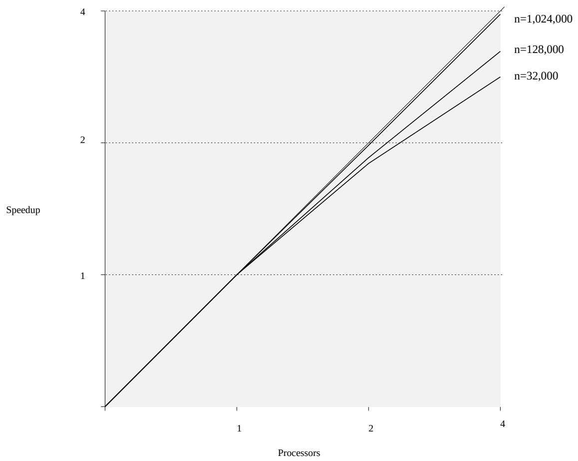

Figure 21: Speedup achieved by algorithm B.4 [high-probability sorting] on a 4 node Silicon Graphics Power Challenge shared memory system.

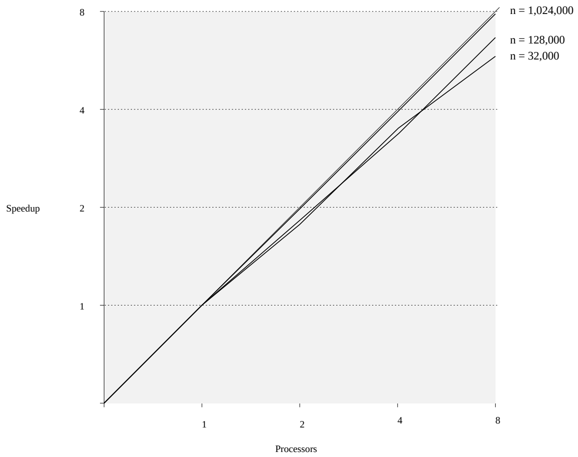

Figure 22: Speedup achieved by algorithm B.4 [high-probability sorting] on an 8 node IBM 9076 SP1 close-coupled distributed memory system.

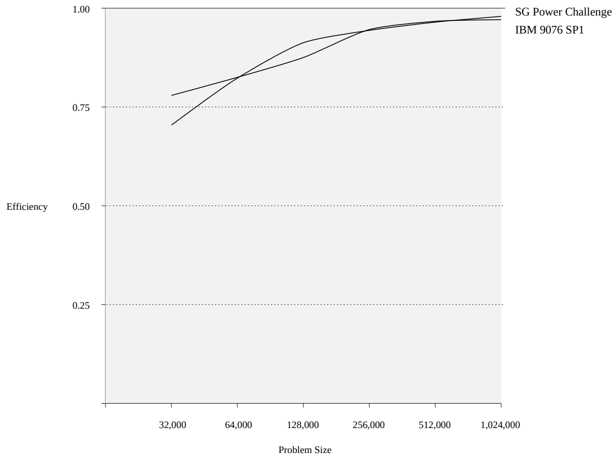

Figure 23: Efficiency of algorithm B.4 [high-probability sorting] on a 4 node Silicon Graphics Power Challenge shared memory system and an 8 node IBM 9076 SP1 close-coupled distributed memory system.

Silicon Graphics Power Challenge							
	Random Data			Real Data			
n	t_{seq}	t_2	t_4	Problem	t_{seq}	t_2	t_4
16K	0.42	0.21	0.11	2Ds16	0.24	0.12	0.07
32K	0.80	0.41	0.22	2Ds32	0.45	0.23	0.12
64K	1.49	0.76	0.39	2Ds64	0.95	0.49	0.25
128K	2.28	1.15	0.58	2Dg128	2.50	1.27	0.64
256K	4.73	2.37	1.18	2Dg256	5.23	2.62	1.31
512K	11.00	5.51	2.75	2Dg512	9.96	4.98	2.49

Table 8: Timing results for algorithm 5.1 [2D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system.

C.3 BSP 2D Convex Hull Computation

In section 5.2.1 we showed how the convex hull vertices of a point set can be identified in optimal expected-time, by a technique known as *gift wrapping*. The worst-case running time of the algorithm was proved to be $O(\frac{n^2}{p})$, which is inferior to the optimal $O(\frac{n \log n}{p})$ bound, however, if it is known in advance that the number of convex hull vertices h is small (i.e. $h = o(\log n)$) then the algorithm defeats the optimal lower bound! For example, it is known that, $h = O(\log n)$ holds for multivariate uniform distributions, $h = O(\sqrt{\log n})$ holds for multivariate normal distributions, and finally $h = O(1)$ holds for multivariate t-distributions drawn from a unit d-cube [60]. In this section we present a BSP implementation (in C) of the algorithm, and analyze experimentally its practical significance.

C.3.1 Performance

The theoretical asymptotic expected performance of the two-dimensional convex hull algorithm is $O(\frac{n \log n}{p})$ for an ample range of values of n, p, g, and L. To determine the practical significance of this complexity bound, we tested our algorithm on two different machines with a varying number of nodes and input sizes. The results for 6 sample problems are presented in tables 8 and 9. The random input consists of random two-dimensional vectors uniformly distributed inside a square.

Figures 24 and 25 present the optimal and actual speedup of algorithm 5.1 for 6 sample problems of size ranging from 16,000 to 512,000 vectors on a Silicon Graphics Power Challenge shared memory system and an IBM 9076 SP1 close-coupled distributed memory system respectively. The nearly linear speedup is evident in both figures.

The relative efficiencies of the 4 node SG Power Challenge implementation and the 8 node IBM 9076 SP1 implementation are compared (graphically) in figure 26. The nearly optimal efficiency is evident in both cases (for sufficiently large problem sizes).

For completeness, we have also compared the practical behaviour of the $O(n \log n)$ expected-time sequential algorithm employed in our implementation with the well-known $O(n \log n)$ worst-case time Graham scan algorithm. Both algorithms have

IBM 9076 SP1									
	Random Data				Real Data				
n	t_{seq}	t_2	t_4	t_8	Problem	t_{seq}	t_2	t_4	t_8
16K	0.53	0.29	0.18	0.16	`2Ds16`	0.31	0.17	0.13	0.07
32K	1.01	0.53	0.30	0.20	`2Ds32`	0.58	0.31	0.17	0.13
64K	1.86	0.96	0.51	0.31	`2Ds64`	1.23	0.63	0.34	0.20
128K	2.82	1.44	0.74	0.42	`2Dg128`	3.20	1.63	0.84	0.46
256K	5.31	2.72	1.39	0.75	`2Dg256`	5.98	3.07	1.56	0.86
512K	11.77	6.02	3.01	1.60	`2Dg512`	10.79	5.56	2.79	1.47

Table 9: Timing results for algorithm 5.1 [2D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system.

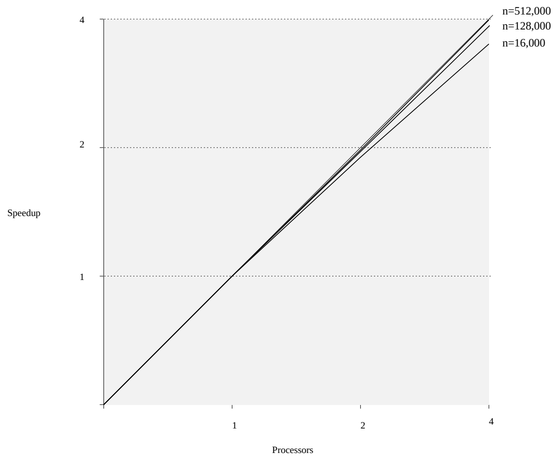

Figure 24: Speedup achieved by algorithm 5.1 [2D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system (real data).

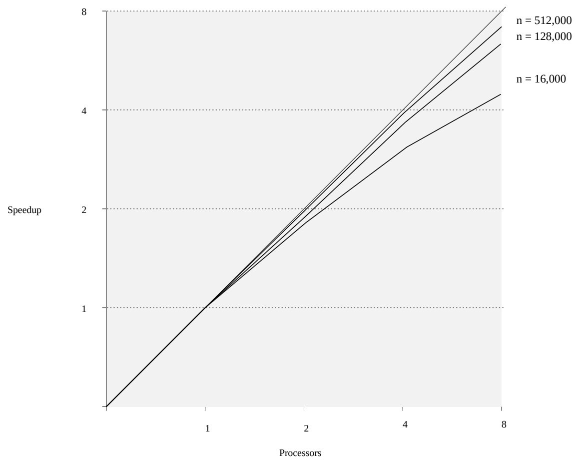

Figure 25: Speedup achieved by algorithm 5.1 [2D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system (real data).

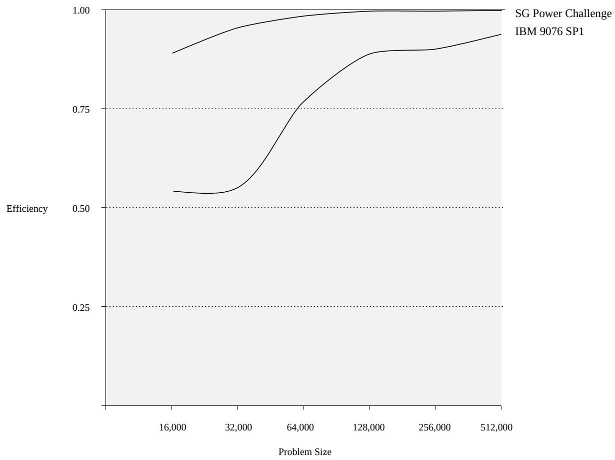

Figure 26: Efficiency of algorithm 5.1 [2D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system and an 8 node IBM 9076 SP1 close-coupled distributed memory system (real data).

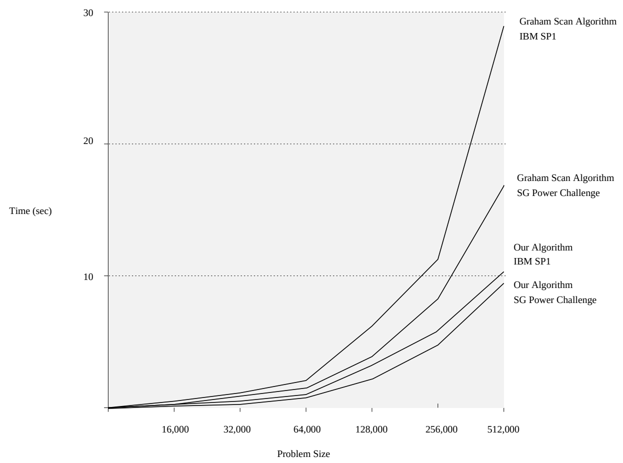

Figure 27: Comparison of sequential convex hull algorithms.

been implemented as C programs on a Silicon Grpahics Power Challenge system and an IBM 9076 SP1 system. We have tried to put the same efforts into both programs to make the comparison as fair as possible. Figure 27 shows the computation times of the algorithms for computing the convex hull of 6 real sample problems.

C.4 BSP 3D Convex Hull Computation

In section 5.3 we presented a high-probability optimal $O(k\frac{n}{p}\log\frac{n}{p})$ time BSP algorithm for computing the convex hull of n points in E^3, for some constant k, under the assumption that the points are drawn independently from any identically distributed continuous function. Examples of distributions that satisfy the above condition include the multivariate normal distribution and multivariate uniform distribution drawn from a unit d-cube [23]. Furthermore, the algorithm was shown to be portable to higher dimensions with no modifications at all! Note that, all adequately documented deterministic parallel algorithms proposed for the general three-dimensional convex hull problem use time at least $O(\frac{n\log^2 n}{p})$. In this section we present a BSP implementation (in C) of the algorithm, and analyze experimentally its practical significance for random and real input distributions. Surprisingly, the algorithm seems to excel on both random and real data.

C.4.1 Performance

The theoretical asymptotic performance of the three-dimensional convex hull algorithm is fundamentally $O(T_{seq}(\frac{n}{p}))$ (with high-probability) for an ample range of values of n, p, g, and L, where $T_{seq}(\frac{n}{p})$ is the time complexity of the underlying sequential algorithm for a problem of size $\frac{n}{p}$. To determine the practical significance of this complexity bound, we tested our algorithm on two different machines with a varying number of nodes and input sizes. The results for 6 sample problems are

Silicon Graphics Power Challenge							
	Random Data			Real Data			
n	t_{seq}	t_2	t_4	Problem	t_{seq}	t_2	t_4
0.4K	0.18	0.09	0.07	`3Dc0.4`	0.12	0.06	0.04
0.8K	0.56	0.25	0.16	`3Dc0.8`	0.40	0.14	0.08
1.6K	1.47	0.55	0.30	`3Dc1.6`	1.13	0.43	0.21
3.2K	4.77	1.75	0.73	`3Dm3.2`	3.49	1.08	0.40
6.4K	16.09	5.27	1.92	`3Dc6.4`	11.52	3.57	1.24
12.8K	51.15	16.72	5.85	`3Dc12.8`	41.83	11.75	3.53

Table 10: Timing results for algorithm 5.3 [3D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system.

IBM 9076 SP1									
	Random Data				Real Data				
n	t_{seq}	t_2	t_4	t_8	Problem	t_{seq}	t_2	t_4	t_8
0.4K	0.22	0.10	0.09	0.13	`3Dc0.4`	0.16	0.07	0.05	0.08
0.8K	0.60	0.30	0.21	0.17	`3Dc0.8`	0.42	0.16	0.09	0.08
1.6K	1.72	0.52	0.33	0.25	`3Dc1.6`	1.36	0.47	0.25	0.21
3.2K	5.80	1.97	0.77	0.54	`3Dm3.2`	4.47	1.32	0.44	0.24
6.4K	20.60	6.33	2.09	1.12	`3Dc6.4`	15.48	4.57	1.45	0.57
12.8K	69.85	21.07	6.93	2.92	`3Dc12.8`	58.34	15.90	4.55	1.56

Table 11: Timing results for algorithm 5.3 [3D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system.

presented in tables 10 and 11.[63] Note that the random input consists of random three-dimensional vectors uniformly distributed inside a cube.

Figures 28 and 29 present the optimal and actual speedup of algorithm 5.3 [3D convex hull] for 6 sample problems of size ranging from 400 to 12,800 vectors on a Silicon Graphics Power Challenge shared memory system and an IBM 9076 SP1 close-coupled distributed memory system respectively. The super-linear speedup is evident in both figures.

[Note that the super-linear speedup is due to the probabilistic nature of the algorithm. It should be a simple exercise to implement a similar probabilistic sequential algorithm that delivers comparable performance.]

C.5 BSP Convex Hull Heuristics

In section 5.4 we presented a simple, rather natural technique (preprocessing step), that is likely to enhance the performance of any convex hull algorithm. The preprocessing step resembles to one step of Quickhull [63], that is, it identifies and eliminates roughly all the points internal to the hull, under suitable assumptions. The algorithm was shown to be remarkably simple and highly efficient.

[63]The sequential 3D convex hull is computed using an $O(n^2)$ incremental convex hull algorithm.

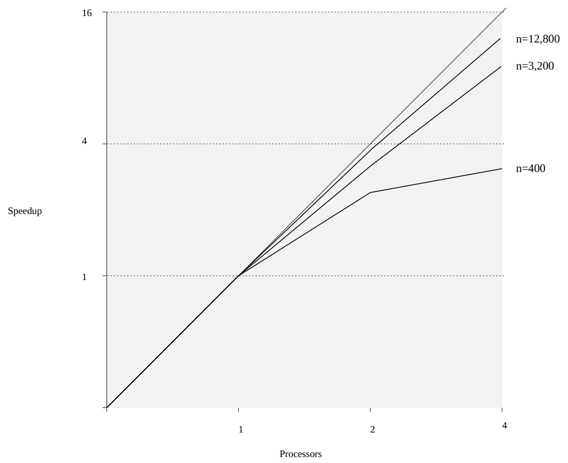

Figure 28: Speedup achieved by algorithm 5.3 [3D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system (real data).

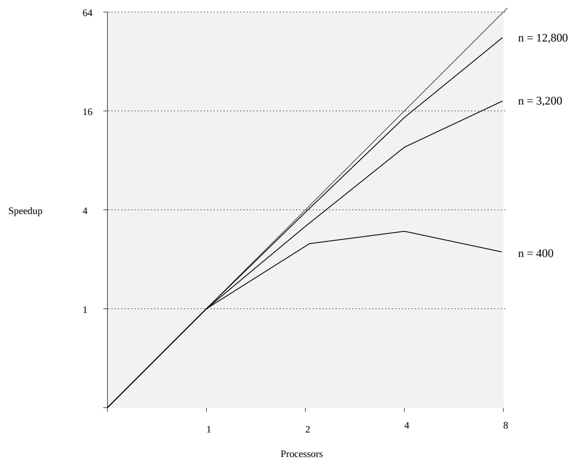

Figure 29: Speedup achieved by algorithm 5.3 [3D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system (real data).

Silicon Graphics Power Challenge					
		Random Data		Real Data	
		Standard	Improved	Standard	Improved
Problem	n	t_4	t_4	t_4	t_4
`2Ds16`	16K	0.12	0.02	0.07	0.02
`2Ds32`	32K	0.22	0.03	0.12	0.02
`2Ds64`	64K	0.39	0.03	0.25	0.03
`2Dg128`	128K	0.58	0.05	0.64	0.05
`2Dg256`	256K	1.18	0.08	1.31	0.10
`2Dg512`	512K	2.75	0.17	2.49	0.19

Table 12: Timing results for algorithm 5.1 [2D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system (with and without pre-processing).

We start by recalling the idea of the two-dimensional algorithm. Suppose that the input vector set S contains n two-dimensional vectors. Pick any four points from S. These points are the vertices of a quadrilateral Q (in the degenerate case Q is a triangle). By its definition Q is contained within the convex hull of S, thus, we can eliminate all points that are inside Q. The goal of the algorithm is to find the four points such that after the elimination step only a few will on average remain. Clearly, the process should be computationally simple and inexpensive, otherwise it is not warranted. In [79], it is shown that a simple and highly efficient choice is the selection of the four points (z_1, z_2, z_3, z_4) that minimize the four functions $(x + y, -x + y, x - y, -x - y)$ (to secure uniqueness choose the points with the lowest indices that minimize these functions). The last step is to eliminate those points that are internal to the smallest rectangle R defined by the horizontal and vertical lines passing through (z_1, z_2, z_3, z_4). If the n points are distributed in the unit d-cube, the above procedure is expected to eliminate all but $O(n^{\frac{d-1}{d}})$ points. In particular, for the two-dimensional case, the above procedure is expected to eliminate all but $8\sqrt{n}$ points [79].

In this section we present a BSP implementation (in C) of the procedure, and analyze experimentally its practical significance for random and real input distributions. The procedure improves significantly the performance of the two- and three-dimensional convex hull algorithms (see sections 5.2.1 and 5.3) for both random and real input distributions.

C.5.1 Performance

We compare the standard two- and three-dimensional algorithms with the improved two- and three-dimensional algorithms implemented according to section 5.4. Tables 12, 13, 14, and 15, show the computation times of the convex hull algorithms for 12 sample problems of size ranging from 400 to 512,000 vectors.

CPU times for the complete set of instances are shown in figures 30, 31, 32, and 33. These figures clearly visualize that even for small problem sizes the pre-processing step is guaranteed to deliver improved performance. The comparison of the standard algorithm with the improved (pre-processed) one gives a clear picture.

IBM 9076 SP1					
		Random Data		Real Data	
		Standard	Improved	Standard	Improved
Problem	n	t_8	t_8	t_8	t_8
`2Ds16`	16K	0.17	0.07	0.07	0.04
`2Ds32`	32K	0.20	0.09	0.13	0.05
`2Ds64`	64K	0.31	0.08	0.20	0.07
`2Dg128`	128K	0.42	0.09	0.46	0.09
`2Dg256`	256K	0.75	0.11	0.86	0.14
`2Dg512`	512K	1.60	0.20	1.47	0.19

Table 13: Timing results for algorithm 5.1 [2D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system (with and without pre-processing).

Silicon Graphics Power Challenge					
		Random Data		Real Data	
		Standard	Improved	Standard	Improved
Problem	n	t_4	t_4	t_4	t_4
`3Dc0.4`	0.4K	0.07	0.07	0.04	0.04
`3Dc0.8`	0.8K	0.16	0.13	0.08	0.07
`3Dc1.6`	1.6K	0.30	0.19	0.21	0.13
`3Dm3.2`	3.2K	0.73	0.35	0.40	0.27
`3Dc6.4`	6.4K	1.92	0.75	1.24	0.60
`3Dc12.8`	12.8K	5.85	1.13	3.53	1.15

Table 14: Timing results for algorithm 5.3 [3D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system (with and without pre-processing).

IBM 9076 SP1					
		Random Data		Real Data	
		Standard	Improved	Standard	Improved
Problem	n	t_8	t_8	t_8	t_8
`3Dc0.4`	0.4K	0.14	0.15	0.08	0.08
`3Dc0.8`	0.8K	0.17	0.16	0.08	0.09
`3Dc1.6`	1.6K	0.25	0.20	0.21	0.15
`3Dm3.2`	3.2K	0.54	0.38	0.24	0.21
`3Dc6.4`	6.4K	1.12	0.66	0.57	0.36
`3Dc12.8`	12.8K	2.92	1.07	1.56	0.62

Table 15: Timing results for algorithm 5.3 [3D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system (with and without pre-processing).

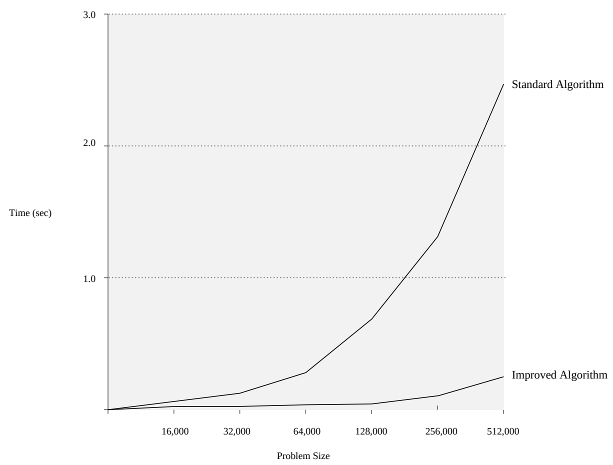

Figure 30: Time achieved by algorithm 5.1 [2D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system (with and without pre-processing on real data).

The improved algorithm is much faster than the standard algorithm for nearly all problem instances. When considering large problems, the improved algorithms seem to exhibit linear running times (even for the three-dimensional case).

References

[1] Abelson, H. "Lower bounds on information transfer in distributed systems." *Journal of the ACM,* 27:384-392, 1980.

[2] Adamowicz, M. and Albano, A. "A two-stage solution of the cutting stock problem." In *Information Processing 1971 Proceedings,* IFIP Congress, pp. 1086-1091, 1971.

[3] Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaig, C., and Yap, C. "Parallel computational geometry." In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science,* pp. 468-477, 1985.

[4] Aggarwal, A. and Huang, M-D.A. "Network complexity of sorting and graph problems and simulating CRCW PRAMS by interconnection networks." In *Proceedings of the 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures,* volume 319 of *Lecture Notes in Computer Science,* (Reif, J., ed.), Springer-Verlag, pp. 339-350, July 1988.

[5] Aggarwal, A., Chandra, A.K., and Snir, M. "Communication complexity of PRAMs." *Theoretical Computer Science,* 71:3-28, 1990.

[6] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.

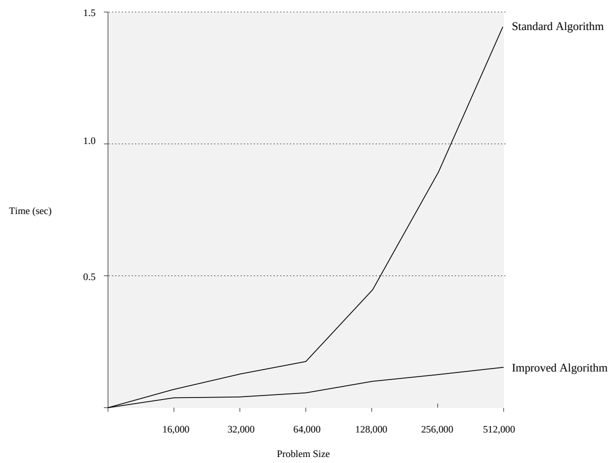

Figure 31: Time achieved by algorithm 5.1 [2D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system (with and without pre-processing on real data).

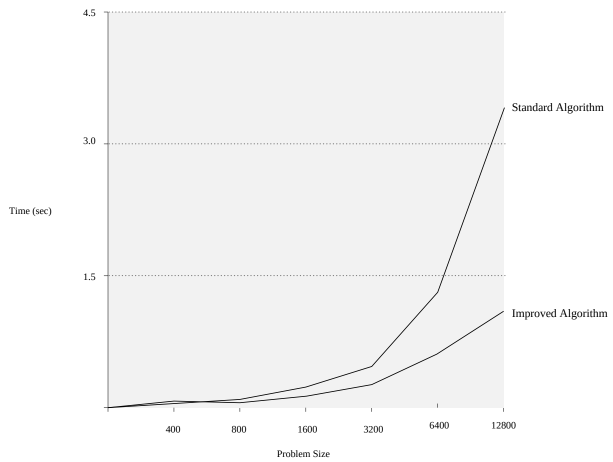

Figure 32: Time achieved by algorithm 5.3 [3D convex hull] on a 4 node Silicon Graphics Power Challenge shared memory system (with and without pre-processing on real data).

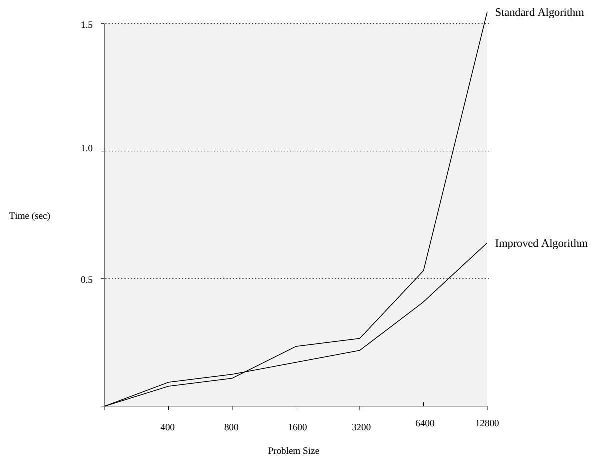

Figure 33: Time achieved by algorithm 5.3 [3D convex hull] on an 8 node IBM 9076 SP1 close-coupled distributed memory system (with and without pre-processing on real data).

[7] AHO, A.V., ULLMAN, J.D., AND YANNAKIS, M. "On notions of information transfer in VLSI circuits." In *Proceeding of the 15th Annual ACM Symposium on Theory of Computing,* pp. 133-139, 1983.

[8] AKL, S.G. "A constant-time parallel algorithm for computing convex hulls." *BIT,* 22:130-134, 1982.

[9] AKL, S.G. "Optimal parallel algorithms for computing convex hulls and for sorting." *Computing,* 33:1-11, 1984.

[10] AKL, S.G. *Parallel Sorting Algorithms.* Academic Press, Florida, 1985.

[11] AKL, S.G. AND GUENTHER, G.R. "Broadcasting with selective reduction." *Proceedings of the 11th IFIP Congress,* San Francisco, pp. 515-520, 1986.

[12] AKL, S.G. "On the power of concurrent memory access." In *Proceedings of the International Conference on Computing and Information,* (Janicki, R. and Koczkodaj, W.W., eds.), Elsevier, pp. 49-55, 1989.

[13] AKL, S.G. AND LYONS, K.A. *Parallel Computational Geometry.* Prentice Hall, New Jersey, 1993.

[14] ANDERSON, R.J. AND MILLER, G.L. "Optical communication for pointer based algorithms." Technical Report CRI 88-14, University of Southern California, 1988.

[15] ASANO, T., EDAHIRO, M., IMAI, H., AND IRI, M. "Practical use of bucketing techniques in computational geometry." *Computational Geometry,* (Toussaint, G.T., ed.), Elsevier Science Publishers B.V., North Holland, 1985.

[16] Assano, T. and Umeo, H. "Systolic algorithms for computing the visibility polygon and triangulation of a polygonal region." *Parallel Computing,* 6:209-216, 1988.

[17] Atallah, M.J. and Goodrich, M.T. "Efficient plane sweeping in parallel." In *Proceedings of the 2nd Annual ACM Symposium on Computational Geometry,* New York, pp. 216-225, 1986.

[18] Atallah, M.J., Cole, R., and Goodrich, M.T. "Cascading divide-and-conquer: A technique for designing parallel algorithms." In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science,* pp. 151-160, 1987.

[19] Atallah, M.J. and Chen, D.Z. "An optimal parallel algorithm for the visibility of a simple polygon from a point." In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry,* Saarbruchen, pp. 114-123, 1989.

[20] Atallah, M.J. and Tsay, J-J. "On the parallel-decomposability of geometric problems." In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry,* Saarbruchen, pp. 104-113, 1989.

[21] Atallah, M.J., Chen, D.Z., and Wagener, H. "An optimal parallel algorithm for the visibility of a simple polygon from a point." *Journal of the ACM,* 38(3):516-533, 1991.

[22] Batcher, K. "Sorting Networks and their applications." In *Proceedings of the AFIPS Spring Joint Computing Conference,* pp. 307-314, 1968.

[23] Bentley, J.L., Kung, H.T., Scholnick, M., and Thomson, C.D. "On the average number of maxima in a set of vectors and applications." *Journal of the ACM,* 25(4):536-543, October 1978.

[24] Bentley, J.L., Weide, B.W., and Yao, A.C. "Optimal expected-time algorithms for closest point problems." *ACM Transactions on Mathematical Software,* 6(4):563-580, 1980.

[25] Bentley, J.L. and Wood, D. "An optimal worst case algorithm for reporting intersections of rectangles." *IEEE Transactions on Computers,* C-29:571-576, 1980.

[26] Bentley, J.L., Faust, G.M., and Preparata, F.P. "Approximation algorithms for convex hulls." *Communications of the ACM,* 25:64-68, 1982.

[27] Bisseling, R.H. and McColl, W.F. "Scientific computing on Bulk-Synchronous Parallel Architectures." Preprint 836, Department of Mathematics, University of Utrecht, December 1993.

[28] Blelloch, G.E. and Little, J.J. "Parallel solutions to geometric problems on the scan model of computation." In *Proceedings of the 1988 International Conference on Parallel Processing,* Illinois, pp. 218-222, 1988.

[29] Blelloch, G.E. "Scans as primitive parallel operations." *IEEE Transactions on Computers,* 38(11)1526-1538, 1989.

[30] Blelloch, G.E. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[31] Blelloch, G.E. "Prefix-sums and their applications." Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

[32] Boots, B.N. and Murdoch, D.J. "The spatial arrangement of random Voronoi polygons." *Computers and Geosciences,* 9:351-365, 1983.

[33] Brown, K.Q. "Voronoi diagrams form convex hulls." *Information Processing Letters,* 9:223-228, 1979.

[34] Chazelle, B. "Computational geometry on a systolic chip." *IEEE Transactions on Computers,* C-33(9):774-785, 1984.

[35] Chazelle, B. Guibas L.J., and Lee D.T. "The power of geometric duality." *BIT,* 25:76-90, 1985.

[36] Chow, A.L. *Parallel algorithms for geometric problems.* PhD Thesis, University of Illinois at Urbana-Champaign, 1980.

[37] Chow, A.L. "A parallel algorithm for determining convex hulls of sets of points in two dimensions." In *Proceedings of the 19th Annual Allerton Conference on Communication, Control and Computing,* Illinois, pp. 214-223, 1981.

[38] Cole, R. and Vishkin, U. "Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms." In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing,* pp. 206-219, 1986.

[39] Cole, R. "Parallel Merge Sort." *SIAM Journal on Computing,* 17(4):770-785, 1988.

[40] Cole, R. and Goodrich, M.T. "Optimal parallel algorithms for polygon and point set problems." In *Proceedings of the 4th ACM Symposium on Computational Geometry,* Illinois, pp. 201-210, 1988.

[41] Cole, R., Goodrich, M.T., and O'Dunlaing, C. "Merging free trees in parallel for efficient Voronoi diagram construction." In volume 443 of *Lecture Notes in Computer Science,* (Paterson, M.S., ed.), Springer-Verlag, pp. 432-445, 1990.

[42] Cole, R. and Zajicek, O. "An optimal parallel algorithm for building a data structure for planar point location." *Journal of Parallel and Distributed Computing,* 8:280-285, 1990.

[43] COOK, S.A. "The classification of problems which have fast parallel algorithms." In Proceedings of the Conference on Foundations of Computation Theory, volume 158 of *Lecture Notes in Computer Science,* (Karpinski, M., ed.), Spinger-Verlag, pp. 78-93, 1983.

[44] COOK, S.A. "An overview of computational complexity." *Communications of the ACM,* 26(6):400-408, 1983.

[45] COOK, S.A. "A taxonomy of problems with fast parallel algorithms." *Information and Computation,* 64:2-22, 1985.

[46] CORMEN, H.T., LEISERSON, C.E., AND RIVEST, L.R. *Introduction to Algorithms.* MIT Press, 1990.

[47] CRAIN, I.K. "Monte Carlo simulation of the random Voronoi polygons: preliminary results." *Search,* 3(6):220-221, 1978.

[48] CYPHER, R. AND PLAXTON, C.G. "Deterministic sorting in nearly logarithmic time on the hypercube and related computers." In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing,* Baltimore, pp. 193-203, 1990.

[49] DADOUN, N. AND KIRKPATRICK, D.G. "Parallel construction of subdivision hierarchies." *Journal of Computer and System Sciences,* 39:153-165, 1989.

[50] DEHNE, F. "$O(n^{1/2})$ algorithms for maximal elements and ECDF searching problem on a mesh-connected parallel computer." *Information Processing Letters,* 22:303-306, 1986.

[51] DEHNE, F. "Solving visibility and separability problems on a mesh-of-processors." *The Visual Computer,* 3:356-370, 1988.

[52] DEHNE, F., PHAM, Q.T., AND STOJMENOVIČ, I. "Optimal visibility algorithms for binary images on the hypercube." Technical Report TR-88-27, Computer Science Department, University of Ottawa, Ontario, 1988.

[53] DEHNE, F., SACK, J-R., AND STOJMENOVIČ, I. "A note on determining the 3-dimensional convex hull of a set of points on a mesh of processors." In *Proceedings of the Scandinavian Workshop on Algorithm Design,* volume 318 of *Lecture Notes in Computer Science,* Springer-Verlag, pp. 154-162, 1988.

[54] DEHNE F. AND RAU-CHAPLIN A. "Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry." *Journal of Parallel and Distributed Computing,* 8:367-375, 1990.

[55] DEHNE, F., FABRI, A., AND RAU CHAPLIN, A. "Scalable parallel geometric algorithms for coarse grained multicomputers." In *Proceedings of the 9th ACM Symposium on Computational Geometry,* San Diego, pp. 298-307, 1993.

[56] DEHNE, F., KENYON, C., AND FABRI, A. "Scalable and architecture independent parallel geometric algorithms with high probability optimal time." In *IEEE Symposium on Parallel and Distributed Processing,* pp. 586-593, 1994.

[57] Dehne, F. "Scalable Parallel Computational Geometry." *Parallel and Distributed Computing,* volume 805 of *Lecture Notes in Computer Science,* (Goos, G. and Hartmanis, J., eds.), Spinger-Verlag, pp. 115-119, 1994.

[58] Devroye, L. "A note on finding convex hulls via maximal vectors." *Information Processing Letters,* 11(1):53-56, August 1980.

[59] Devroye, L. and Toussaint, G.T. "A note on linear expected time algorithms for finding convex hulls." *Computing,* 26:361-366, 1981.

[60] Devroye, L. "Expected time analysis of algorithms in computational geometry." *Computational Geometry,* (Toussaint, G.T., ed.), Elsevier Science Publishers B.V., North Holland, 1985.

[61] Dowd, P.W. "High performance interprocessor communication through optical wavelength division multiple access channels." In *Proceedings of the 18th Annual International Symposium on Computer Architecture,* pp. 96-105, 1991.

[62] Dyer, C.R. "A fast parallel algorithm for the closest pair problem." *Information Processing Letters,* 11(1):49-52, 1980.

[63] Eddy, W. "A new convex hull algorithm for planar sets." *ACM Transactions on Mathematical Software,* 3(4):398-403, 1977.

[64] Edelsbrunner, H. and Mucke, E.P. "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms." *ACM Transactions on Graphics,* 9(1):66-104, January 1990.

[65] Emiris, I. and Canny, J. "An efficient approach to removing geometric degeneracies." In *Proceedings of the 8th ACM Symposium on Computational Geometry,* Berlin, pp. 74-82, 1992.

[66] Eppstein, D. and Gallil, Z. "Parallel algorithmic techniques for combinatorial computation." *Annual Review of Computer Science,* 3:233-283, 1988.

[67] Evans, D.J. and Stojmenoviċ, I. "On parallel computation of Voronoi diagrams." *Parallel Computing,* 12:121-125, 1989.

[68] Flanders, H. *Calculus.* W.H. Freeman and Company, New York, 1985.

[69] Fortune, S. and Wyllie, J. "Parallelism in random access machines." In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing,* pp. 114-118, 1978.

[70] Fortune, S. "A sweepline algorithm for Voronoi diagrams." *Algorithmica,* 2:153-174, 1987.

[71] Fortune, S. and van Wyk, C.J. "Efficient exact arithmetic for computational geometry." In *Proceedings of the 9th ACM Symposium on Computational Geometry,* San Diego, pp. 163-172, 1993.

[72] Frazer, W.D. and McKellar, A.C. “Samplesort: A sampling approach to minimal storage tree sorting.” *Journal of the ACM,* 17(3):496-507, July 1970.

[73] Freeman, H. and Shapira, R. “Determining the minimum-area encasing rectangle for an arbitrary closed curve.” *Communications of the ACM,* 18(7):409-413, 1975.

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

[75] Gerbessiotis, A.V. and Siniolakis, C.J. “Deterministic sorting and randomized median finding on the BSP model.” To appear in *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[76] Gereb-Graus, M. and Tsantilas, T. “Efficient optical communication in parallel computers.” In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures,* pp. 41-48, 1992.

[77] Gibbons, A.M. and Rytter, W. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, UK, 1988.

[78] Gibbons, P.B. “Asynchronous PRAM Algorithms.” In *Synthesis of Parallel Algorithms,* Morgan Kaufmann Publishers, San Mateo, 1993.

[79] Golin, M. and Sedgewick, R. “Analysis of a simple yet efficient convex hull algorithm.” in *Proceedings of the 4th ACM Symposium on Computational Geometry,* Illinois, pp. 153-162, 1988.

[80] Goodrich, M.T. “Finding the convex hull of a sorted point set in parallel.” *Information Processing Letters,* 26:173-179, 1987.

[81] Goodrich, M.T. “Intersecting line segments in parallel with an output-sensitive number of processors.” Technical Report 88-27, Department of Computer Science, The John Hopkins University, Baltimore, 1988.

[82] Goodrich, M.T., Shauck, S.B., and Guha, S. “Parallel methods for visibility and shortest path problems in simple polygons.” In *Proceedings of the 6th Annual Symposium on Computational Geometry,* Berkeley, pp. 73-82, 1990.

[83] Gottlieb, A. and Kruskal, C.P. “Complexity results for permuting data and other computations on parallel processors.” *Journal of the ACM,* 13(2):193-209, 1984.

[84] *Grand Challenges: High Performance Computing and Communications.* The FY 1992 U.S. Research and Development Program, A Report by the Committee on Physical, Mathematical, and Engineering Sciences, To Supplement the U.S. President’s Fiscal Year 1992 Budget, 1992.

[85] Green, P.J. and Sibson, R. “Computing Dirichlet tessellations in the plane.” *The Computer Journal,* 21(2):168-173, 1978.

[86] HAIMS, M.J. AND FREEMAN, H.A. "A multistage solution of the template-layout problem." *IEEE Transactions on Systems Science and Cybernetics SSC-6,* 2:145-151, 1970.

[87] HANAN, M. "Layout, interconnection, and placement." *Networks,* 5:85-88, 1975.

[88] HARTMANN, A. AND REDFIELD, S. "Design sketches fro optical crossbar switches intended for large scale parallel processing applications." *Optical Engineering,* 29(3):315-327, 1989.

[89] HERSHBERGER, J. "Finding the upper envelope of n line segments in $O(n \log n)$ time." *Information Processing Letters,* 33:169-174, 1989.

[90] HILLIS, W.D. AND STEELE, G.L. "Data parallel algorithms." *Communications of the ACM,* 29(12):1170-1183, 1986.

[91] HOARE, C.A.R. "Quicksort." *Computer Journal,* 5:10-15, 1962.

[92] HOFFMANN, C. *Geometric and Solid Modelling.* Morgan Kaufmann Publishers, San Mateo, 1989.

[93] HOLEY, J.A. AND IBARRA, O.H. "Iterative algorithms for planar convex hull on mesh-connected arrays." In *Proceedings of the 1990 International Conference on Parallel Processing,* Illinois, pp. 147-150, 1990.

[94] HOLEY, J.A. AND IBARRA, O.H. "Triangulation, Voronoi diagram, and convex hull in k-space on mesh-connected arrays and hypercubes." In *Proceedings of the 1991 International Conference on Parallel Processing,* Illinois, pp. 147-150, 1991.

[95] JÁJÁ, J. AND KUMAR, P. "Information transfer in distributed computing with applications to VLSI." *Journal of the ACM,* 31:150- 162, 1984.

[96] JÁJÁ, J. *An Introduction to Parallel Algorithms.* Addison- Wesley, 1992.

[97] JARVIS, R.A. "On the identification of the convex hull of a finite set of points in the plane." *Information Processing Letters,* 2:18-21, 1973.

[98] JEONG, C-S. AND LEE, D.T. "Parallel geometric algorithms on a mesh-connected computer." *Algorithmica,* 5(2):155-178, 1990.

[99] JEONG, C-S. "An improved parallel algorithm for construction Voronoi diagram on a mesh-connected computer." *Parallel Computing,* 17:505-514, 1991.

[100] JEONG, C-S. "Parallel Voronoi diagram in L_1 (L_∞) on a mesh-connected computer." *Parallel Computing,* 17:241-252, 1991.

[101] KARP, R.M. AND RAMACHANDRAN, V. "A survey of parallel algorithms for shared memory machines." Technical Report, Computer Science Division, UC Berkeley, 1988.

[102] KARP, R.M. AND RAMACHANDRAN, V. "Parallel algorithms for shared-memory machines." In *Handbook of Theoretical Computer Science,* (van Leeuwen, J., ed.), North Holland, 1990.

[103] KIRKPATRICK, D.G. "Optimal search in planar subdivisions." *SIAM Journal on Computing,* 12(1):28-35, 1983.

[104] KNUTH, D.E. *The Art of Computer Programming. Volume I: Fundamental Algorithms.* Addison-Wesley, Reading, 1968.

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

[106] KNUTH, D.E. "Big Omicron and big omega and big theta." *SIGACT News*, 8(2):18-24, April 1976.

[107] KUNG, H.T., LUCCIO, F., AND PREPARATA, F.P. "On finding the maxima of a set of vectors." *Journal of the ACM,* 22(4):469-476, October 1975.

[108] LADNER, R.E. AND FISCHER, M.J. "Parallel prefix computation." *Journal of the ACM,* 27(4):831-838, 1980.

[109] LEE, D.T. AND PREPARATA, F.P. "Location of a point in a planar subdivision and its applications." *SIAM Journal on Computing,* 6:594-606, 1977.

[110] LEE, D.T. AND CHEN, I.M. "Display of visible edges of a set of convex polygons." *Computational Geometry,* (Toussaint, G.T., ed.), Elsevier Science Publishers B.V., North Holland, 1985.

[111] LEE, D.T. AND PREPARATA, F.P. "Parallel batched planar point location on the CCC." *Information Processing Letters,* 33:175-179, 1989.

[112] LEIGHTON, F.T. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufman, California, 1991.

[113] LEVCOPOULOS, C., KATAJAINEN, J., AND LINGAS, A. "An optimal expected-time parallel algorithm for Voronoi diagrams." in *Proceedings of the Scandinavian Workshop on Algorithm Theory,* volume 318 of *Lecture Notes in Computer Science,* Springer-Verlag, 190-198, 1988.

[114] LU, M. "Constructing the Voronoi diagram on a mesh-connected computer." In *Proceedings of the 1986 International Conference on Parallel Processing,* Illinois, pp. 806-811, 1986.

[115] MCCOLL, W.F. "General purpose parallel computing." In *Lectures on parallel computation,* (Gibbons, A. and Spyrakis, P., eds.), Cambridge University Press, 1993.

[116] MCCOLL, W.F. "Scalable parallel computing: A grand unified theory and its practical development." In *Proceedings of IFIP World Congress*, 1:539-546, Hamburg, August 1994.

[117] McColl, W.F. "An architecture independent programming model for scalable parallel computing." *Portability and Performance for Parallel Processors,* (Ferrante, J. and Hey, A.J.G., eds.), John Wiley and Sons, 1994.

[118] McEnzie, P.D. and Stout, Q.F. "Asymptotically efficient hypercube algorithms for computational geometry." In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation,* Maryland, pp. 8-11, 1990.

[119] McEnzie, P.D. and Stout, Q.F. "Practical hypercube algorithms for computational geometry." In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation (poster presentation),* Maryland, 1990.

[120] Meisel, W.S. *Computer-Oriented Approaches to Pattern Recognition.* Academic Press, 1972.

[121] Miller, R. and Stout, Q.F. "Computational Geometry on a mesh- connected computer." In *Proceedings of the 1984 International Conference on Parallel Processing,* Michigan, 66-73, 1984.

[122] Miller, R. and Stout, Q.F. "Geometric algorithms for digitized pictures on a mesh-connected computer." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* PAMI-7(2):216-228, 1985.

[123] Miller, R. and Stout, Q.F. "Mesh computer algorithms for digitized pictures." *IEEE Transactions on Computers,* C-38(3):321-340, 1989.

[124] Miller, R. "A library for bulk-synchronous parallel programming." In *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing,* December 93.

[125] Miller, R. and Reed, J.L. "The Oxford BSP library: user's guide. Version 1.0." Oxford Parallel Technical Report, Oxford university Computing Laboratory, 1994.

[126] Liu, C.L. *Introduction to Combinatorial Mathematics.* McGraw-Hill, New York, 1968.

[127] Nath, D., Maheshwari, N., and Bhatt, P.C.P. "Parallel algorithms for the convex hull problem in two dimensions." Technical Report EE 8005, Department of Electrical Engineering, Indian Institute of Technology, India, 1980.

[128] Okabe, A., Boots, B.N., and Sugihara, K. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley and Sons Ltd, New York, 1992.

[129] Overmars, M.H. and van Leeuwen, J. "Maintenance of Configurations in the plane." *Journal of Computer Systems Science,* 23:166-204, 1981.

[130] Papadimitriou, C.H. and Sipser, M. "Communication complexity." *Journal of the ACM,* 28:260-268, 1981.

[131] Papadimitriou, C.H. and Ullman, J.D. "A communication-time trade-off." *SIAM Journal on Computing,* 16:639-647, 1987.

[132] Papadimitriou, C.H. and Yannakis, M. "Towards and architecture-independent analysis of parallel algorithms." In *Proceedings of the 20th Annual Symposium on Theory of Computing,* pp. 510-513, 1988.

[133] Parzen, E. *Modern Probability Theory and Its Applications.* John Wiley and Sons, 1960.

[134] Paturi, R., Lu, D-T., Ford, J.E., Esener, S.C., and Lee, S.H. "Parallel algorithms based on expander graphs for optical computing." *Applied Optics,* 30(8):917-927, 1991.

[135] Pavlidis, T. *Algorithms for Graphics and Image Processing.* Computer Science Press, 1982.

[136] Preilowski, W. and Mumbeck, W. "A time-optimal parallel algorithm for the computing of Voronoi diagrams." In *Proceedings of the Graph-Theoretic Concepts in Computer Science,* volume 344 of *Lecture Notes in Computer Science,* (van Leeuwen, J., ed.), Springer-Verlag, pp. 424-433, 1988.

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

[138] Quine, M.P. and Watson, D.F. "Radial generation of n-dimensional Poisson processes." *Journal of Applied Probability,* 21:548-557, 1984.

[139] Rao, S.B. "Properties of an interconnection architecture based on wavelength division multiplexing." Technical Report TR-92-009-3-0054-2, NEC Research Institute, Princeton, 1992.

[140] Raynaud, H. "Sur l'enveloppe convexe des nuages des points aléatoires dans R^n." *International Journal of Applied Probability,* 7:35-48, 1970.

[141] Reif, H.J. and Valiant L.G. "A logarithmic time sort for linear size networks." *Journal of the ACM,* 34:60-76, January 1987.

[142] Reif, J.H. and Sen, S. "Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications." In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures,* Crete, pp. 327-339, 1990.

[143] Reif, J.H. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, San Mateo, 1993.

[144] Reinelt, G. "The Travelling Salesman." In volume 840 of *Lecture Notes in Computer Science,* Springer-Verlag, pp. 40-41, 1994.

[145] Renyi, A. and Sulanke, R. "Über die konvexe Hülle von n zufällig gewählten Punkten, I." *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete,* 2:75-84, 1963.

[146] RENYI, A. AND SULANKE, R. "Über die konvexe Hülle von n zufällig gewählten Punkten, II." *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete,* 3:138-147, 1964.

[147] RENYI, A. AND SULANKE, R. "Zufällige konvexe Polygone in einem Ringgebiet." *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete,* 9:146-157, 1968.

[148] RHYNSBURGER, D. "Analytic delineation of Thiessen polygons." *Geographical Analysis,* 5:133-144, 1973.

[149] ROES, P.B.M. "A note on linear programming algorithm design: a combinatorial problem." *Communications of the ACM,* 9(5):340-342, May 1966.

[150] ROSENFELD, A. *Picture Processing by Computers.* Academic Press, New York, 1969.

[151] SANZ, J.L.C. *Opportunities and Constraints of Parallel Computing.* Springer-Verlag, 1988.

[152] SAXENA, S., BHATT, P.C.P., AND PRASAD, V.C. "Efficient VLSI parallel algorithm for Delaunay triangulation on orthogonal tree network in two and three dimensions." *IEEE Transactions on Computers,* C-39(3):400-404, 1990.

[153] SAXENA, S., BHATT, P.C.P., AND PRASAD, V.C. Correction to: "Efficient VLSI parallel algorithm for Delaunay triangulation on orthogonal tree network in two and three dimensions." *IEEE Transactions on Computers,* C-40(1):122, 1991.

[154] SHAMOS, M.I. AND HOEY, D. "Closest point problems." *Seventh Annual IEEE Symposium on Foundations of Computer Science,* pp. 208-215, October 1976.

[155] SRIDHAR, R., IYENGAR, S.S., AND RAJANARAYANAN, S. "Range search in parallel using distributed data structures." In *Proceedings of the International Conference on Databases, Parallel Architectures, and Their Applications,* Miami, pp. 14-19, 1990.

[156] STOJMENOVIC̀, I. "Computational geometry on a hypercube." In *Proceedings of the 1988 International Conference on Parallel Processing,* Illinois, pp. 100-103, 1988.

[157] STOUT, Q.F. "Pyramid computer solutions of the closest pair problem," *Journal of Algorithms,* 6:200-212, 1985.

[158] STOUT, Q.F. "Constant-time geometry on PRAMS." In *Proceedings of the 1988 International Conference on Parallel Processing,* Illinois, pp. 104-107, 1988.

[159] TAMASSIA, R. AND VITTER, J.S. "Optimal cooperative search in fractional cascaded data structures." In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures,* Crete, pp. 307-316, 1990.

[160] TAMASSIA, R. AND VITTER, J.S. "Planar transitive closure and point location in planar structures." *SIAM Journal on Computing,* 20(4):708-725, 1991.

[161] THOMPSON, C.D. "Area-time complexity for VLSI." In *Proceedings of the 11th Annual Symposium on Theory of Computing,* pp. 81-88, 1979.

[162] TIWARI, P. "Lower bounds on communication complexity in distributed computer networks." *Journal of the ACM,* 34:921-938.

[163] TOUSSAINT, G.T. *Computational Geometry.* Elsevier Science Publishers B.V., North Holland, 1985.

[164] VALIANT, L.G. "Parallelism in comparison problems." *SIAM Journal on Computing,* 4:348-355, 1975.

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

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

[167] VALIANT, L.G. "A combining mechanism for parallel computers." Parallel Architectures and Their Efficient Use, in *Proceedings of the First Heinx Nixdorf Symposium,* Padeborn, November 1992.

[168] VISKIN, U. "Structural parallel algorithmics." In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming,* in volume 510 of *Lecture Notes in Computer Science,* (Leach Albert, J., Monien, B., and Rodriguez, A., eds.), Springer-Verlag, pp. 363-380, 1991.

[169] VORONOI, G. "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième Mémoire: Recherches sur les parallélloèdres primitimfs." *Journal für die Reine und Angewandte Mathematik,* 134:198-287, 1908.

[170] WANG, B-F. AND CHEN, G-H. "Constant time algorithms for sorting and computing convex hulls." Technical Report, Department of Computer Science and Information Engineering, National Taiwan University, Taiwan, 1990.

[171] WEE, Y.C. AND CHAIKEN, S. "An optimal parallel L_1-metric Voronoi diagram algorithm." In *Proceedings of the 2nd Canadian Conference on Computational Geometry,* Ontario, pp. 60-65, 1990.

[172] WIRTH, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[173] WYLLIE, J.C. "Communication structures for large networks of microcomputers." *IEEE Transactions on Computers,* C-30(4):264-273, 1981.

[174] WYLLIE, J.C. *The complexity of parallel computations.* PhD Thesis, Department of Computer Science, Cornell University, 1981.

[175] YAO, A.C. "Some complexity questions related to distributed computing." In *Proceedings of the 11th Annual Symposium on Theory of Computing,* pp. 324-332, 1979.

[176] YAO, A.C. "A lower bound to finding convex hulls." *Journal of the ACM,* 28(4):780-787, 1980.

[177] YAP, C.K. "What can be parallelized in computational geometry," In *Parallel Algorithms and Architectures,* volume 269 of *Lecture Notes in Computer Science,* (Albrecht, A., Jung, H., and Mehlhorn, K., eds.), Springer-Verlag, pp.184-195, 1987.