Collection Types for Database Programming in the BSP Model

K Ronald Sujithan and Jonathan M D Hill
Programming Research Group
Oxford University Computing Laboratory
{Ronald.Sujithan,Jonathan.Hill}@comlab.ox.ac.uk

Abstract

We study the pragmatics of integrating collection types, that model a broad class of non-numerical applications, into the Bulk Synchronous Parallel (BSP) model which abstracts a diversity of parallel architectures using just four numerical parameters. We outline how the collection types have been built on-top of the direct BSP programming environment provided by *BSPlib*, give results on a SGI PowerChallenge and IBM SP2, and discuss how these types can help implement object databases.

1 Introduction

Parallel computers have been successfully deployed in many scientific and numerical application areas, but their use in commercial applications, which are often non-numerical in nature, has been scarce. One of the impediments in the long-term commercial uptake of parallel computing has been the proliferation of differing machine architectures and corresponding programming models. However, due to several technological and economic reasons, the various classes of parallel computers such as shared-memory machines, distributed-memory machines, and networks of workstations are beginning to acquire a familiar appearance: a workstation-like processor-memory pair at the node level, and a fast, robust interconnection network that provides node-to-node communication [11].

The aim of recent research into the Bulk Synchronous Parallel Model [12,20] has been to take advantage of this architectural convergence. The central idea of BSP is to provide a high level abstraction of parallel computing hardware whilst providing a realisation of a parallel programming model that enables architecture independent programs to deliver scalable performance on diverse hardware platforms. Although BSP programming environments have been successfully utilised in several numerical applications, they provide little help in non-numeric applications as they are based upon an explicit distribution and manipulation of arrays among a number of processors.

Previous approaches to adding collection types to parallel languages were taken from a specific programming model viewpoint. Data-parallel approaches achieve parallelism by the wholesale manipulation of collection types such as arrays, sets, or lists. A general unifying approach to data-parallelism is exemplified by the work on data-parallel functional programming [8] in terms of the Bird-Meertens Formalism or Skeletons [3,4,16]. However, apart from a plethora of purely theoretical results, there have been few practical implementations of data-parallel functional languages other than the language NESL developed by Blelloch [1].

The goal of this paper is to study the cost-effective realisation of a number of well known *collection types*, in order to provide higher-level programming abstractions within a BSP programming environment. Specifically, the work described in this paper considers the pragmatics of

supporting the functional style of data-parallelism in the BSP model. We focus on the problem of uniting data-parallelism, data distribution and load balancing and accurate cost prediction within the BSP model. Our work is motivated by an object database case study, and the type system needed to model a broad class of database applications. We provide some encouraging results from a preliminary BSP implementation of the **select-from-where** form of a database query.

The rest of this paper is organised as follows. We conclude this section with an outline of the application case study. In section 2, we introduce the BSP model, performance prediction under the model, and the *BSPlib* programming environment. Section 3 introduces the *BSP-Collections* model of sets, lists, and bags, and section 4 present the BSP implementation of collections and a variety of operations on collections. Finally, in section 6 we provide results from the case study.

1.1 The case study

The Object Database Management Group (ODMG), as part of the OMG Organisation (an industrial consortium established to promote object technology) has proposed an industry standard for object databases [2]. The ODMG-93 standard defines an object model of data, and a companion object definition (ODL) and query languages (OQL). We work with the main elements of the data model, particularly the data types supported — the generalised collections data type and the specialised data types *sets, bags, lists* and *arrays*— and the operations defined for these types [17].

The utility of the collections datatypes considered in this paper will be demonstrated in the context of supporting elements of ODMG. Our example is essentially a database query based upon this standard, which forms an ideal, representative, non-numerical application, to evaluate the programming concepts under consideration.

2 The BSP model

Before collection types are considered, we give a brief introduction to the BSP model. A *BSP computer* is an abstraction of any real parallel computer, and can be decomposed into three parts:

(1) a number of processor/memory components
(usually each consists of sequential processor with a block of local memory);
(2) an interconnection network that delivers messages in a point-to-point manner;
(3) a facility for globally synchronising *all* the processors by means of a barrier.

The *BSP model* defines the way in which programs are executed on the BSP computer. The model defines that a program consists of a sequence of *supersteps*. During a superstep, each processor-memory pair can perform a number of local computations on values held locally to each processors memory at the start of the superstep. During the superstep, each processor may also initiate communication with other processor-memory pairs. However, the model does not prescribe any particular style of communication, although it does require that at the end of a superstep there is a barrier synchronisation at which point any pending communications must be completed [19].

2.1 BSP cost model

For each of the algorithms we describe later, we analyse their computational and communication costs. The superstep methodology that forms the heart of the BSP model facilities cost analysis because the cost of a BSP program running on a number of processors is simply the sum of

each separate superstep executed by the program. In turn, for each superstep, the costs can be decomposed into those attributed to purely local computation, global exchange of data, and barrier synchronisation. To ensure that cost analysis can be performed in an *architecture independent* way, cost formulas are parameterised by four *architecture dependent* constants:

p	number of processor/memory pairs
s	computation speed of a single processor (#flops)
l	minimum time for barrier synchronisation (#flops)
g	the ratio of local computational operations/second to the #words delivered by the network

Given these four constants, then the cost of a superstep is captured by the formulae $\alpha + \beta g + l$ [6]; where α is an *architecture independent cost* that models the maximum number of operations executed by *one* of the processes in the local computation phase of a superstep; and β is the largest accumulated size of all messages either entering or leaving a process within a superstep.

2.2 The Oxford BSP toolset

Two modes of BSP programming are possible: (1) in *automatic mode* [20] the run-time system hides memory distribution/management from the user (i.e., PRAM style of programming); (2) in *direct mode* the programmer retains control over data distribution and manipulation among the processors. The aim of this paper can be considered to provide an automatic mode for a variety of collection types on-top of a library of direct mode primitives outlined next.

A number of researchers are currently forming a World-Wide Standard BSP Library [7] by synthesising several low-level (direct mode) BSP programming approaches that have been pursued over the last few years. They propose a library called *BSPlib* to provide a parallel communication library based around a SPMD model of computation.

The main parts of *BSPlib* are: (1) routines to spawn a number of processes to be run in an SPMD manner; (2) an operation that synchronises all processes, and therefore identifies the end of one superstep and the start of the next; (3) Direct Remote Memory Access (DRMA) communication facilities that allow a process to manipulate the address space of remote processes *without the active participation of the remote process*; (4) Bulk Synchronous Message Passing (BSMP) operations that provide a non-blocking send operation that delivers messages to a system buffer associated with the destination process at the end of a superstep.

The results described in this paper have been obtained using the Oxford BSP toolset implementation of *BSPlib* [9], which also contains a profiling tool to help visualise the inter-process communication patterns occurring in each superstep. The profiling tool graphically exposes three important pieces of information: (a) the elapsed time taken to perform communication; (b) the pattern of communication; (c) the computational elapsed time. When discussing the results in section 6, we will highlight these three attributes.

3 BSP realisation of collections

We consider three particular kinds of collections, with different applications in mind: (1) *sets* are unordered collections that contains no duplicates; (2) *bags* are unordered collections that allow duplicates; (3) *lists* (or sequences) are ordered collections that allow duplicates; Sets and bags are fundamental types in database schema design, whereas lists play a dual role. Firstly lists act as a fundamental construct that represent several useful non-linear data structures, such as graphs,

trees etc., in object databases. Secondly, we use sorted lists without duplicates as the central type in our parallel implementation.

A collection $\mathit{collection}\langle\tau\rangle$ containing n elements of type τ is initially distributed among p processors of a BSP machine in blocks of n/p elements (other distribution schemes such as, whole, cyclic and random, are also possible). We maintain the data-structure invariant using a parallel sorting algorithm, to be described next, which works upon an extra tag that we associate with each element of type τ in the collection $\mathit{collection}\langle\tau\rangle$. For a list based collection, the tag will be a simple boolean valued attribute that determines if a collection element is "active". However, for a set based collection the tag is the element itself of type τ, which ensures the data-structure invariance of sorted lists.

Sorting Several sorting methods have been devised for a variety of parallel architectures, including BSP sorting algorithms [6]. After careful consideration of these methods, we have selected and implemented a practical variant of the sample sort algorithm [13,15]. Central to this method is the fact that runs of elements containing the same key will be distributed to the same processor – this property is utilised in the algorithms that follow.

Our implementation, (1) performs a local sort; (2) each processor selects p regular samples which are gathered onto processor zero; (3) these samples are sorted and p regular pivots are picked from the p^2 samples; (4) these pivot values are broadcast to all the processors; (5) each processor partitions its local block using the pivots; (6) each processor sends it's partition i to processor i; and finally (7) local merge of the sorted partitions produces the desired effect of a global sort.

Since the complexity of an optimal comparison-based sequential sort is $\textsc{SeqSort}(n) = n \log n$, then the cost of locally sorting p processors chunks of data is $\textsc{SeqSort}(n/p)$. The communication cost is gn/p for the permute (i.e., at most the entire segment needs to be transferred to a different processor), and therefore the upper-bound BSP cost for the parallel sort is $\textsc{ParSort}(n) = \textsc{SeqSort}(n/p) + gn/p + l$ (refined analysis, including the load balancing properties of this algorithm, can be found in [18]).

4 Collection operations

We first describe the fundamental collection operations expressed as the familiar higher-order functions (i.e. functions that take a function as an argument), *map*, *filter*, and *fold*. The BSP implementation of each of these operations is presented, in the context of the distributed collection types introduced above, along with their BSP costs. Where appropriate, we indicate alternative algorithms for these operations based on the parameters, c, p, l and g, where c is the maximum cost of applying the function f to a single element of the collection. We mention the database interpretations of these higher-order functions in section 7.

4.1 Single collection operations

Map We begin with the most basic collection operator *map* that applies a function $f : \tau \to \sigma$ to every element of a collection $\mathit{collection}\langle\tau\rangle$ to create a new collection $\mathit{collection}\langle\sigma\rangle$. It is the essence of the data-parallel paradigm as it models embarressingly parallel computations because the application of f to each element in the collection can occur concurrently. If we ignore nesting, then the BSP cost is cn/p for this operation.

Fold Fold (or reduce) is a higher order function that given an associative and commutative (commutativity may be required to work with unordered set collections) operator $\otimes$ and a collection $[x_1, x_2, \ldots, x_n]$ computes the values $x_1 \otimes x_2 \otimes \ldots \otimes x_n$. The parallel interpretation of fold is that a sequential fold can be performed in parallel to each segment of the collection of size n/p in time proportional to cn/p. Depending on the relative values of c, p, l and g, each of the p results from the local computation can then be obtained by using either, (1) a divide-and-conquer technique to fold the p values held in each process in $\log p$ supersteps, or, (2) in just a single superstep. The BSP cost of the entire fold operation will be, (1) $cn/p + \log p\ (l + g + c)$, and, (2) $cn/p + cp + pg + l$, repectively.

Filter Given a collection $collection\langle\tau\rangle$, then a filtered version of the collection that satisfies a predicate $f : \tau \to \mathbb{B}$. In the context of collections distributed among p processors in n/p chunks, a global filter can be implemented in terms of p local filters with cost cn/p, but there are potential problems with data-skew as the filter may produce a new collection that is significantly smaller than the original. This potential skewing of data can be addressed by a rebalancing algorithm based on the parallel prefix technique [6]. Alternatively, the load balancing properties of the sorting routine can be employed *en route*, as demonstrated in our simulations (section 5).

4.2 Multiple collection operations

By using the type conversion rules discussed in [18], or using the ODMG-93 explicitly defined function **listtoset** for this purpose, any collection type can be converted into a *set* collection. Sets can therefore provide the basis for our implementation of multiple collection operations, without resorting to tailored implementations of each collection type.

Our task therefore reduces to one of implementing all the set-theoretic operations on set collections. As pointed out in [14], a fast sorting procedure can be used as the basis for the efficient processing of these operations. Given two sets S and T, we describe how to calculate the set theoretic operations $S - T$ (difference), $S \cup T$ (union), and $S \cap T$ (intersection) in BSP. We make a similar assumption to those in section 3 about the distribution of S and T; that is n elements of S and m elements of T are distributed across the p processors.

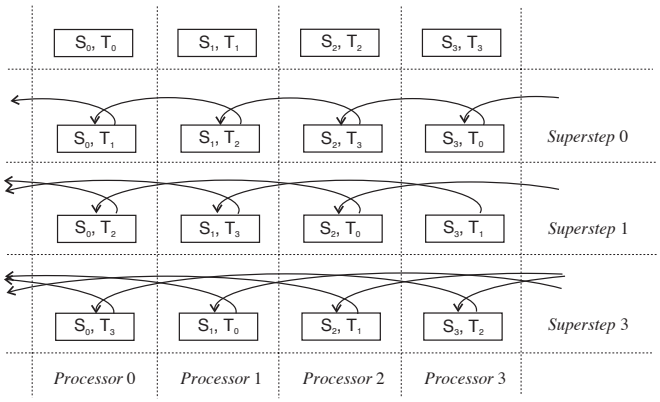

Figure 1: The cartesian product algorithm

Cartesian product Given two collections S of type $collection\langle\tau\rangle$ and T of type $collection\langle\sigma\rangle$, then the cartesian product $S \times T$ calculates the collection of all pairs from S and T of type

collection$\langle\tau\times\sigma\rangle$. The BSP implementation works by transposing the data blocks of the smaller collection, say T, in successive supersteps, and then computing the local cartesian product. Clearly, this will bring every element s_i of S with every element t_i of T into the same processor, to form the pair (s_i, t_i) within $p-1$ supersteps. The BSP cost of local computation is cnm/p^2 and communication gm/p, thus the total cost is (assuming $m \leq n$), $cmn/p^2 + gm/p + l$ for a single iteration. Thus the total cost is $cmn/p + (p-1)(gm/p + l)$ for all $p-1$ supersteps. Figure 1 shows this algorithm for 4-processors.

Difference To compute $S - T$ in BSP, we: (1) locally append the n/p and m/p list elements of the sorted list implementation of the sets together; (2) locally sort the appended list; (3) locally eliminate duplicates, as duplicates are only introduced when an element from T is in S; (4) perform a global sort on the appended lists (as the sorting algorithm we use never causes sequences of similar valued elements to straddle processors); then (5) a final local elimination phase removes duplicates, which completes the implementation of set difference. The BSP cost of the algorithm is $c(n+m)/p + \textsc{SeqSort}((n+m)/p) + \textsc{ParSort}(n+m)$.

Union and Intersection Union is implemented in a manner similar to set difference. That is, to compute $S \cup T$: (1) the local blocks of S and T are appended, sorted, and then all-but-one duplicates eliminated; (2) a global sort is performed; (3) as sort does not cause "runs" to straddle processors, locally eliminating all-but-one duplicate completes set union. The BSP cost of this algorithm is also $(n+m)/p + \textsc{SeqSort}((n+m)/p) + \textsc{ParSort}(n+m)$. Using the same assumptions, $S \cap T$ can be implemented in terms of $S - (S - T)$.

5 An object database query

We synthesise the basic operations discussed so far into a case study of an object database query according to the ODMG-93 definition. The essence of the ODMG-93 standard is to extend the set-based relational model with collection types, and OQL which allows high-level expression of database query statements. As [10] observes, OQL is essentially an extension of SQL to deal with collection-valued fields instead of just set-valued fields. The most common form of a query in OQL is a select statement of the form:

select e
from x_1 **in** C_1, x_2 **in** C_2, ..., x_n **in** C_n
where *pred*

Such statements translate into the following comprehension notation [5] $bag\{\ e \mid x_1 \leftarrow C_1,\ x_2 \leftarrow C_2,\ \ldots,\ C_n \leftarrow C_n,\ pred\ \}$. Processing this query involves applying the filter predicate (i.e. selection) to the collections involved (to reduce the number of elements), taking the cartesian product (i.e. join) of the (filtered) collections, $C_1 \times C_2 \times \cdots \times C_n$, followed by a map function that selects the attributes specified by the expression e (i.e. projection).

We have simulated this OQL **select-from-where** construct using the *BSPlib*, and analysed the results using the Oxford BSP Toolset. Our simulation method is described next. Two collections C_1 and C_2 of 1024 keys each are defined to represent a database, and after the two collections are filtered, they are individually sorted to regain order and load balance. A cartesian product is formed next, which is followed by sorting and duplicate elimination. Finally the map function selects the desired attributes. The collection sizes we chose are small enough so that communication latencies are paramount in comparison to local computation.

6 Experimental results

Machine	p	s	l	g
Uniprocessor Sun	4	0.7	198000	68
SGI PowerChallenge	4	74	1902	9.3
IBM SP2 (switch)	8	26	5412	11.4

Table 1: BSP parameters

In this section, results are presented for our BSP simulation on a uniprocessor Sun workstation, a 4-processor shared-memory SGI Power Challenge, and an 8-processor distributed-memory (switch based) IBM SP/2. Table 1 shows the observed BSP parameter values (in #flops) for these machines, and profiling charts generated from execution of the simulation on each machine are discussed in the context of these values. They highlight the salient features of processing collections on real machines[1].

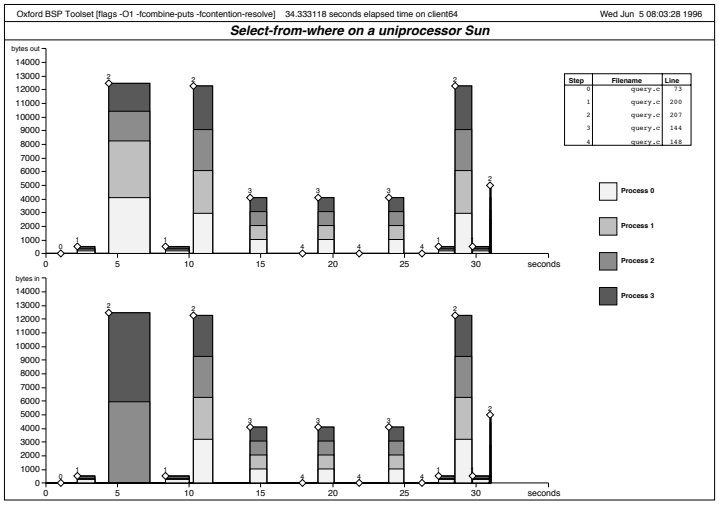

Figure 2: Sun Workstation profile

In the profiling charts, such as the one shown in figure 2, the top graph contains bars whose height identifies the amount of data leaving each process, whereas the lower graph shows the amount of data entering each processor during the superstep. The width of the bars indicate the elapsed time taken to perform communication and synchronisation within a superstep, whereas the "white spaces" or gaps between the bars indicate local computation within the specified superstep.

Figure 2 shows the profiling results from a Sun workstation. The bars marked with 1 relate to the communication involved in gathering the regular samples and then redistributing the pivot values to all the processors during the (sample) sorting phase, as described in section 4. The bars marked with 2 relate to the communication intensive data redistribution phase of sorting. Since this phase also restores load balance in our simulation, it could be viewed as the communication involved in load balancing intermediate results.

[1]These simulations are run using the same program, without any machine specific optimisations, and using the native *BSPlib* implementation.

The $p-1=3$ bars marked with 3 relate to the communication involved during the computation of cartesian product. As described in section 4 the algorithm uses $p-1$ supersteps to permute the data blocks, to bring the blocks into the appropriate processors. The gaps in between the bars represent the time for computing the local cartesian product. Other gaps between the bars relate to the compute-bound map that performs projection and filter which performs the selection.

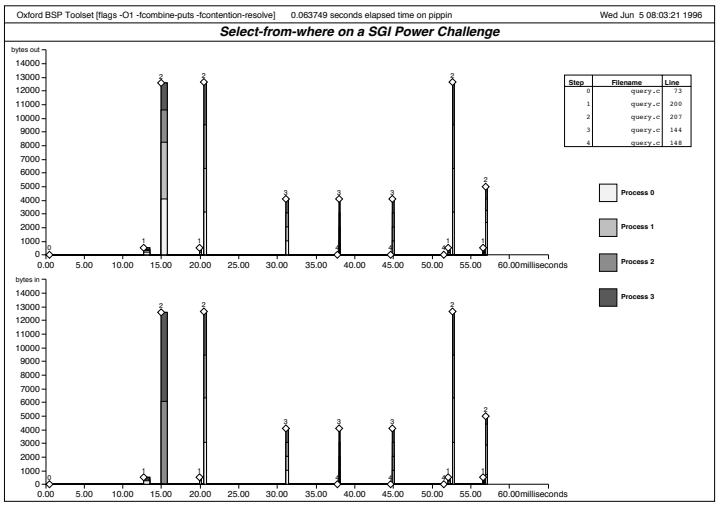

Figure 3: SGI PowerChallenge profile

In contrast, figure 3 shows the execution of the same program on a 4-processor SGI Power Challenge. Apart from the much reduced time for communication due to a much reduced value for g on an PowerChallenge (note the change of scale from seconds to milliseconds), processors also perform more balanced communication, although they exchange the same amount of information. The substantial reduction in the gap between the bars reflects the speedup obtained using four (faster) processors.

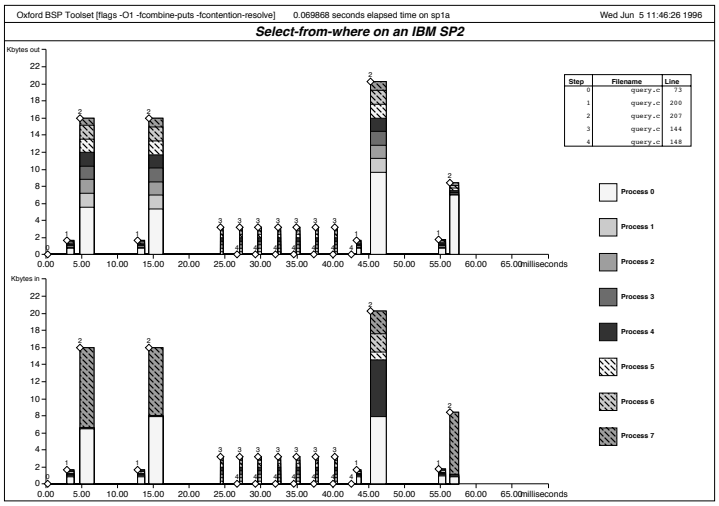

Figure 4: IBM SP2 profile

In figure 4, results are shown for the execution of the simulation on an 8-processor IBM SP/2. In this case, $p-1=7$ supersteps are needed to complete the cartesian product, as the bars

marked with 3 indicate, but the amount of data communicated during each superstep is reduced correspondingly. The slight difference in the width of the bars is due to the difference in the rate of communication on a distributed memory machine as compared to a shared memory based implementation, that is, the SP/2 has a higher value of g than the SGI. However the smaller gaps between the bars indicates the additional speedup attained due to the availability of extra processors in the embarrassingly parallel operations.

7 Conclusions

In this paper, we have investigated a way to provide high-level programming abstractions in terms of collection types for parallel database programming. In particular, the task of incorporating a representative group of collection types, that can model a broad class of non-numerical applications, into the BSP programming environment was considered.

We briefly introduced the ongoing work on *BSPlib*, and this paper has taken the first steps towards providing an automatic BSP programming environment, that provides a sufficient level of abstraction to tackle the challenges of non-numerical computations. Databases and database query processing, remains to be a challenging application in this class, and can benefit from the progress being made in general-purpose parallel computing. Finally, we have outlined how a BSP environment augmented with richer types can help to implement object database operations. Our experimental simulation on parallel machines show encouraging results for further work.

References

[1] G. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, Department of Computer Science, Carnegie Mellon University, April 1993.

[2] R. G. G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Francisco, California, 1994.

[3] M. Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. In G. Joubert, editor, *Parallel Computing: Trends and Applications*, pages 489–492. North-Holland, 1994.

[4] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming using Skeleton Functions. In *Parallel Architectures and Languages Europe*, 1993.

[5] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *ACM SIGMOD International Conference on Management of Data*, May 1995.

[6] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[7] M. W. Goudreau, J. M. D. Hill, K. Lang, W. F. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A Proposal for the BSP Worldwide Standard Library (Preliminary Version). Technical report, Oxford University Computing Laboratory, Apr. 1996. (see `www.bsp-worldwide.org` for more details).

[8] J. M. D. Hill. *Data-Parallel Lazy Functional Programming.* PhD thesis, Dept of Computer Science, Queen Mary & Westfield College, University of London, Sept. 1994.

[9] J. M. D. Hill, P. I. Crumpton, and D. A. Burgess. Theory, Practice, and a Tool for BSP Performance Prediction. In *EuroPar'96*, number 1124 in LNCS. Springer-Verlag, Aug. 1996.

[10] W. Kim. Observations on the ODMG-93 Proposal for an Object-Oriented Database Language. *SIGMOD Record*, 23(1):4–9, 1994.

[11] W. F. McColl. General Purpose Parallel Computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, pages 337–391. Cambridge University Press, 1993.

[12] W. F. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in LNCS, pages 46–61. Springer-Verlag, 1995.

[13] J. H. Reif and L. G. Valiant. Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, 34(1):60–76, 1987.

[14] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, Oct. 1980.

[15] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[16] D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[17] K. R. Sujithan. An Object Model of Data, Based on the ODMG Industry Standard for Database Applications. In *IEE/BCS International Seminar on Client/Server Computing*, Oct. 1995.

[18] K. R. Sujithan. Towards a Scalable Parallel Object Database – The Bulk Synchronous Parallel Approach. Technical Report PRG-TR-17-96, Oxford University Computing Laboratory, Aug. 1996.

[19] L. G. Valiant. Bulk Synchronous Parallel Computers. In M. Reeve and S. E. Zenith, editors, *Parallel Processing and Artificial Intelligence*, pages 15–22. Wiley, Chichester, U.K., 1989.

[20] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.