BSP-Like External-Memory Computation*

Jop F. Sibeyn† Michael Kaufmann‡

Abstract

In this paper we present a paradigm for solving external-memory problems, and illustrate it by algorithms for matrix multiplication, sorting, list ranking, transitive closure and FFT. Our paradigm is based on the use of BSP algorithms. The correspondence is almost perfect, and especially the notion of x-optimality carries over to algorithms designed according to our paradigm.

The advantages of the approach are similar to the advantages of BSP algorithms for parallel computing: scalability, portability, predictability. The performance measure here is the total work, not only the number of I/O operations as in previous approaches. The predicted performances are therefore more useful for practical applications.

1 Introduction

Sequential Computation. The von Neumann model (RAM model) is strongly established, and the availability of a generally accepted model has allowed for tremendous progress in the theory of algorithms. The basic assumption of this model is that all stored data can be accessed in unit-time. This is not an absolute truth, not even in current practice: data in cache can be accessed at least ten times faster than data that are not in cache.

In the future, with ever-increasing amounts of data, this assumption will break down even further (see [3] for a detailed description). Yet, most programmers accept the RAM model without discussion, and may try to exploit some cache-features in a final optimization. But, even now we can encounter sequential problems that should not be dealt with on the basis of the von Neumann model. On a typical work-station, accessing a datum in main memory is several thousand times faster than accessing a datum that is stored on the hard-disc. The effect of this cannot be neglected, and thus special algorithms are necessary for solving problems that are so large that their data do not fit into the main memory: *external-memory algorithms*. There are only a few researchers who have intensively considered this field [1, 5, 4, 13].

Parallel Computation and the BSP Model. The situation in parallel computation is different: the lack of a unifying model is viewed as one of the main reasons why progress in this domain has been so much slower than in sequential computation. Here, numerous models exist, and a lot of energy is still invested in model discussions, and the wheel has been reinvented several times. There is an enormous gap between most theoretical models and the practice of parallel machines.

In 1990 Valiant proposed the BSP (Bulk Synchronous Parallel) model to bridge this gap and to provide the community with a possible unifying model. In a recent paper [12], McColl gives a

*Part of this research was carried out while the second author was visiting the Max-Planck-Institut.

†Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. E-mail: jopsi@mpi-sb.mpg.de.

‡Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13, 72076 Tübingen, Germany. E-mail: mk@informatik.uni-tuebingen.de

detailed overview of the current state of BSP. He states that the great strength of the model lies in its

- scalability,
- portability,
- predictability.

Each of these features is clearly of crucial importance. Of course, the model also has its drawbacks, most importantly, some programming flexibility is sacrificed for uniformity, which may lead to less efficient algorithms for a particular parallel machine or problem. This need not be taken too seriously: above we pointed out that in sequential computation, for the sake of a simple and universal model, cache effects are neglected without much discussion. Indeed the BSP model has become quite popular in the past few years (see the proceedings of Euro-Par '96, LNCS 1123, to find at least ten papers dealing with theoretical and practical aspects of the model and its modifications).

In the BSP model, the performance of a parallel computer is characterized by only three parameters:

p: the number of processors;

g: the total number of local operations performed by all processors in one second, divided by the total number of words delivered by the communication network in one second in a situation of continuous traffic.

l: the number of time steps required for performing a barrier synchronization;

In the model, a parallel computation is subdivided in supersteps, at the end of which a barrier synchronization and a routing is performed. Hereafter all requests for data that were posted during a preceding superstep are fulfilled. We consider the cost of a superstep. If w is the maximum number of internal operations in a superstep, and h is the maximum number of words sent or received by any processing unit, *PU*, then the number of time steps $T_{\text{superstep}}$ to perform it, is given by [12]

$$T_{\text{superstep}}(w, h) = w + h \cdot g + l. \tag{1}$$

The parameter l can be interpreted to take the start-up latency into account, and may depend on the diameter of the network. g takes care of the throughput of the network: the larger it is, the weaker the network. For a true PRAM, $l = g = 0$. More details on the BSP model are given in [18, 11, 12]. An extension is discussed in [2].

External-Memory Computation. All modern operating systems provide a mechanism for managing virtual memory. This allows large problems to be solved on systems with moderate primary (RAM) memory. Part of the data actually resides in main memory, while the rest is paged-out into the secondary memory. The secondary memory might be any medium for bulk-storage that allows reads and writes, but we suppose it to be a hard-disc.

If a piece of data is requested that is not stored in the main memory, then its *page* (a block of data with contiguous indices) is swapped with some page that has so far resided in main memory. This causes two additional forms of delay: rotational delay, for the disk-head to reach the position of the header of the page, and transfer delay, for the page to be loaded into main-memory. Both delays are far larger than the time for accessing a datum in main memory. Therefore it is very important to design algorithms that take external-memory effects into account. Unfortunately, as in parallel computation, in the field of external-memory algorithms a common widely-acknowledged model of computation is missing.

This Paper. We introduce a paradigm that may help to find good external-memory algorithms as the result of a guided search through algorithms in the already much further developed field of parallel algorithms. We start with some examples that set the direction for this search. Then we consider the limitations of an earlier proposal.

In Section 4, we propose the paradigm itself. We prove that the quality measure of BSP algorithms carries over to algorithms designed according to our paradigm. This feature fundamentally distinguishes our approach from anything before: parallel algorithms serve not only as a source of inspiration, but by well-established analytical means the most suitable candidate can be selected. This we consider to be our main result. The paper concludes with some examples, illustrating how to handle the paradigm.

2 Guiding Examples

We introduce the problematic of external-memory computation by comparing the execution time of two trivial programs. We argue that the first models the access pattern of well-structured operations, and the second that of chaotic operations. Then we present a third approach, which models the access pattern that arises when the strategy of this paper is applied.

2.1 Structured Versus Random Access

Consider the following, almost identical programs:

Program A;
var *row*: **array** $[0 \,..\, N-1]$ **of int** ;
function *rand*: select random number from $\{0, 1, \ldots, N-1\}$;
(**for** $i := 0$ **to** $N-1$ **do**
row$[i]$:= *rand*) .

Program B;
var *row*: **array** $[0 \,..\, N-1]$ **of int** ;
function *rand*: select random number from $\{0, 1, \ldots, N-1\}$;
(**for** $i := 0$ **to** $N-1$ **do**
row[*rand*] := i) .

The only difference is that in Program A a random number is assigned to consecutive positions of the array, while in Program B consecutive numbers are attributed to random positions of the array.

The computational effort of both programs is the same, but the memory is accessed differently. The precise timing of these programs depends on the computer under consideration and its operating system. In order to provide some numbers, we implemented them in C on a SPARC10 workstation running under Solaris with a 32MB main memory (of which about 22MB are freely available) and a 80MB swap space. The resulting time consumptions are given in Table 1. Program C will be described in Section 2.2.

We see that the time consumptions for Program A and B increase linearly until $N = 5 \cdot 10^6$. In this range, the difference in their time consumptions is probably due to cache effects. We see that it is more or less correct to neglect them: even though a number in cache can be accessed ten

N	T_A/N	T_B/N	T_C/N
$2 \cdot 10^6$	2.13	3.58	3.54
$4 \cdot 10^6$	2.14	3.58	3.54
$5 \cdot 10^6$	2.19	4.54	3.78
$6 \cdot 10^6$	2.26	1071	3.86
$8 \cdot 10^6$	2.30	3525	3.93
$12 \cdot 10^6$	2.43	7314	4.07
$16 \cdot 10^6$	2.40	12563	4.11

Table 1: The time consumptions for the programs A, B and C in microseconds, divided by the number of entries of the array N.

times faster than a number not in cache, the effect on the performance of the whole program is only about 50%.

For $N > 5 \cdot 10^6$, the array does not fit into the main memory anymore (an integer is 4 bytes long, and a few MB are used by the system). For Program A the time consumption increases only 15% because of this.[1] For Program B, however, in which the memory access is chaotic, such that almost every step means a page fault, the time consumption explodes. The time again becomes linear, but the number of operations performed per second is almost 4000 times smaller than before! It comes down to less than 80 passes of the loop per second.

In the remainder of the text we use the following notation:

$$\begin{aligned} M &= \textit{memory-size} \\ N &= \textit{problem-size} \\ B &= \textit{page-size} \end{aligned}$$

Here one should be consistent, and specify all numbers in terms of bytes or in terms of integers (four bytes). We refer to bytes.

Program A models the access pattern of a well-written program operating on an array or a matrix, for which we may assume that the elements of the list are handled in order. If $N > M$, then the number of page faults is only

$$pf_A = (N - M)/B$$

Program B models the access pattern of a program operating on a list or a graph. In this case, the access pattern cannot be structured by the programmer, and will in general be chaotic. Thus, for $N > M$, the expected number of page faults is given by

$$pf_B = (N - M/B) \cdot (1 - M/N).$$

For large N, the difference with pf_A approximates B. Thus, with typical page-sizes around 8KB, pf_B may be several thousand times larger than pf_A. As for trivial programs, like Program A and B, the time is determined by how fast the data can be loaded, which results in a similar factor between the time consumptions.

[1] From another experiment, we know that computation and fetching missing pages overlap to a large extent. Thus, from the results of Program A, one should not conclude that paging costs only 0.27s per million integers. The actual paging time is about 2s per million integers, which amounts to 2MB / s.

2.2 Blocked Random Access

Program A and B are extreme cases. The following program has a mixed structure:

> **Program** C;
> **var** *row*: **array** $[1 .. N]$ **of int** ;
> **function** *rand*: select random number from $\{0, 1, \ldots, m-1\}$;
> (**for** $i := 0$ **to** $N/m - 1$ **do**
>   **for** $j := 1$ **to** m **do**
> $row[i \cdot m + rand] := j$) .

Again the accessed row position is chosen randomly, but in this case at any time only from a block of size m. In our implementation we chose $m = 10^6$.

The time consumptions for various values of N are given in Table 1. For $N < M$, we see that Program C has approximately the same time consumption as Program B. This is not surprising: as in Program B, cache-faults are very common in Program C.[2] On the other hand, for $N > M$, Program B behaves dramatically differently from Program C: just as Program A, it slows down only by 15%. This is not surprising either: as long as $m < M$, the number of page faults in Program C is approximately the same as in Program A. Namely, if this condition holds, then all pages that are relevant to a block of size m are loaded into the memory once, then accessed for a while, and finally replaced.

3 PRAM Algorithms?

In Section 4, we present a paradigm which allows us to perform external-memory algorithms with essentially the memory-access pattern of Program C. This reduces the time consumption from that of Program B to that of Program C. The paradigm is based on the observation that under certain conditions simulating parallel algorithms leads to good external-memory algorithms.

Already Chiang et al. [4] proposed simulating parallel algorithms for external-memory computation. They suggested simulating PRAM algorithms. In such a sequential simulation, the data have to be paged-in for every communication step. With some examples we illustrate that this may be problematic, and does not easily lead to good external-memory algorithms.

Example 1 *Consider multiplying two $n \times n$ matrices in parallel. The standard PRAM algorithm, see for example [7], uses $n^3/\log n$ PUs and solves the problem in $\mathcal{O}(\log n)$ time. Simulating this sequentially would require $\mathcal{O}(n^3/\log n)$ memory, which is excessive (considering that n is very large).*

Taking some more care, we could simulate a PRAM algorithm with $P = 3 \cdot n^2/M$ PUs. Assuming that $P \leq n$, an obvious PRAM algorithm would be to let every PU multiply $n/P = M/(3 \cdot n)$ rows of A with all bundles of n/P columns of B. The choice of P is inspired by the fact that now all data of a PU, plus the computed results, just fit into the main memory. Simulating this requires that all elements of B are paged-in $P = \Theta(n^2/M)$ times.

Alternatively, we could multiply all $\sqrt{M/3} \times \sqrt{M/3}$ submatrices of A and B, and add the results together. This algorithm needs to page all data into the main memory only $\mathcal{O}(n/\sqrt{M})$ times, which is the square root of the above.

[2] Indeed, if we set $m = 10{,}000$, then the time per million items drops to 2.58, which comes close to the result for Program A. However, in the light of the much more important effect of paging, we will not try to optimize cache behavior as well.

In this extremely simple case the choice of the right algorithm is clear, but only indirectly: knowing the optimal external-memory algorithm, we see that this is indeed also a good PRAM algorithm.

In Example 1, the choice of the best external-memory algorithm is not obvious, but at least it goes back on an optimal strategy for multiplying matrices on PRAMs (by recursive two-division). The situation may be even worse. Our following example shows that in some cases the best external-memory algorithm is not (a modification of) an optimal PRAM algorithm with a reduced number of PUs.

Example 2 *Consider sorting N numbers. Suppose that $M < N \leq M^{3/2}/2$ (the example can be generalized for larger N). A good sorting strategy would be to apply column-sort [10] or a variant thereof. This requires that all data are paged-in only a constant number of times. As a PRAM algorithm, column-sort is not very interesting: it does not lead to a $\mathcal{O}(\log N)$ algorithm.*

A comparably good external-memory algorithm is to sort all subsets of size M, and then to apply an N/M-way merge. As a PRAM algorithm this does not make sense, this is a typical sequential approach.

A much better PRAM algorithm is based on repeated two-way merging. On a PRAM, two best algorithm has work $\mathcal{O}(N \cdot \log N)$, and time $\mathcal{O}(\log N)$ [6]. As an external-memory algorithm, it requires that all data are paged in at least $\mathcal{O}((\log(N/M))$ times.

In general, for deriving external-memory algorithms, one must bring along a considerable understanding of the problem, to choose the best of several possible PRAM algorithms, and sometimes even look elsewhere. This is far from a mechanical process. PRAM algorithms might serve as a source of inspiration, but only in a very loose sense. The fundamental problem is that algorithms that are indistinguishable as PRAM algorithms, from the point of view of their complexity, may perform very differently when simulated sequentially. The principal cause is that in PRAM algorithms communication is free, and thus PRAM algorithms may comprise a large number of communication steps. In other words, the model has been oversimplified, missing certain essential characteristics.

4 Paradigm

4.1 BSP-Like Algorithms

We are motivated by the good performance of Program C. Each of the operations on a chunk of size m can be viewed as the operations of a PU on its internal memory. In our case, we perceive Program C as the simulation of a virtual parallel machine with $P = N/m$ PUs. Of course, so far these PUs operate in isolation, but the communication in a parallel machine can be modeled by writing 'messages' into a $P \times P$ matrix. Our paradigm is a formalization of these observations:

Paradigm 1 *Divide the memory space into P blocks of suitable size. Then develop a BSP-like algorithm for a virtual machine with P processors and execute it sequentially.*

More concretely, for a problem $\mathcal{P}$ of size $N > M$, we suggest performing the following steps:

1. *Set $m = M/3$.*

2. *Suitably divide the memory space of size N into $P = \lceil N/m \rceil$ blocks $B_0, \ldots, B_{P-1}$, of size m at most.*

3. *Design a* BSP-like algorithm *for $\mathcal{P}$, for a virtual machine with P PUs.*

4. *Execute the BSP-like algorithm on the sequential computer.*

We still have to specify Step 3 and Step 4:

Definition 1 *A BSP-like algorithm is an algorithm that proceeds in discrete* supersteps*, and satisfies the following conditions:*

- *In Superstep s, $s \geq 1$, PU_i, $0 \leq i < P$, operates only on the data in the block B_i and on the messages $Mes(j, i, s)$, $0 \leq j \leq P - 1$.*
- *In Superstep s, $s \geq 1$, PU_i, $0 \leq i < P$, generates messages $Mes(i, j, s + 1)$ to be 'sent' to PU_j, $0 \leq j \leq P - 1$. The size of $Mes(i, j, s + 1)$ is at most m/P.*
- *The initial messages, $Mes(i, j, 1)$, $0 \leq i, j < P$ are void.*

Lemma 1 *If a BSP-like algorithm is executed on a parallel computer, then no PU ever has to store more than M data at the same time.*

Proof: In any Superstep s, PU_i must store at most the data in block B_i, plus the data in the messages $Mes(j, i, s)$ and $Mes(i, j, s+1)$, $0 \leq j < P$. In all, these are at most $m + 2 \cdot P \cdot (m/P) = 3 \cdot m = M$ data. □

Lemma 1 implies that BSP-like algorithms are indeed suitable for execution on a sequential computer with a main-memory of size at least M:

Program SEQUENTIAL_EXECUTION
for $s \geq 1$ **do**
  for $0 \leq i < P$ **do**
    Page-in B_i and $Mes(j, i, s)$, $0 \leq j < P$.
    Perform the operations of PU_i in Superstep s.
    Page-out B_i and $Mes(i, j, s + 1)$, $0 \leq j < P$.

In the last step, for $s \geq 2$, $Mes(i, j, s + 1)$ may overwrite $Mes(i, j, s - 1)$ to save storage. This gives

Lemma 2 *A BSP-like algorithm can be executed on a sequential computer with main-memory size M and storage capacity $3 \cdot N$, with $3 \cdot M/B$ paging operations per superstep.*

If one is willing to program at the level of the operating system, then such explicit context switches as occur in SEQUENTIAL_EXECUTION might be performed several times faster than just leaving the paging to the standard pager.

4.2 Relation to BSP-Algorithms

An external-memory algorithm that works in accordance with the paradigm is called BSP-like, because, with a suitable definition of the parameters, it is a direct simulation of a parallel BSP algorithm. As we already implicitly assumed: *In our cost estimates, we only consider paging-in operations.*

Corresponding to the parameters (p, g, l), we have (P, G, L):

$$\begin{aligned} P &= \lceil 3 \cdot N/M \rceil, \\ G &= \frac{\#\{\text{internal operations per second}\}}{\#\{\text{words that can be paged-in per second}\}}, \\ L &= M \cdot G/3. \end{aligned}$$

Notice that G is *not* a big number: it takes many steps to page-in a whole page of size B, but a page contains many words as well. For the computer plus hard-disc on our desk, $G \simeq 2 \cdot 10^6 / 4 \cdot 10^5 = 5$. The definition of G is completely general, and does not presuppose that there is only one hard-disc.

With these definitions, the correspondence with the BSP model is almost perfect. Let W_s be the total number of operations performed during Superstep s, and let

$$H_s = \sum_{0 \leq i,j < P} \mathit{Size}\{\mathit{Mes}(j, i, s)\},$$

then we can express the cost of a superstep as concisely as in (1):

Theorem 1 *The number of time steps $T_{\mathit{superstep}}$, to perform Superstep s, is given by*

$$T_{\mathit{superstep}}(W_s, H_s) = W_s + H_s \cdot G + P \cdot L.$$

Proof: Paging-in all data takes $\sum_i (B_i + \sum_j \mathit{Size}\{\mathit{Mes}(j, i, s)\}) \cdot G = P \cdot M/3 \cdot G + H_s \cdot G$. Carrying out the operations themselves takes W_s steps. □

Let S be the number of performed supersteps. Then we get the following expression for the total number of time steps T_{ext}, for solving the problem:

$$T_{\text{ext}}(P, G, L) = \sum_{s=1}^{S} (W_s + H_s \cdot G + P \cdot L). \tag{2}$$

The first term is due to computation, the last two to 'communication'. Considerations that apply to the BSP model can be repeated here. It also leads to our main theorem, which shows that the choice of the parameters is correct:

Theorem 2 *Consider an external-memory problem of size N, with parameters (P, G, L). Suppose that for parameters $(p, g, l) = (P, G, L)$ there is a BSP algorithm in which no PU ever stores more than N/p data, running in $T_{\mathit{par}}(p, g, l)$ time steps. Let $T_{\mathit{ext}}(N, M)$ be the number of time steps required for the corresponding BSP-like algorithm. Then,*

$$T_{\mathit{ext}}(N, M) \leq p \cdot T_{\mathit{par}}(p, g, l).$$

Proof: Let S be the number of performed supersteps, then we know from (1), that

$$T_{\text{par}}(p, g, l) = \sum_{s=1}^{S} (w_s + h_s \cdot g + l).$$

Here w_s is the maximal work any PU has to perform in Superstep s, and likewise is h_s the maximum number of packets any PU has to send or receive at the end of Superstep s. Thus, for the corresponding BSP-like algorithm, $W_s \leq p \cdot w_s$ and $H_s \leq p \cdot h_s$. Combining these facts with (2) completes the proof. □

4.3 Quality Measure

From the theory of BSP algorithms, we also adopt the following quality measure:

Definition 2 *An external-memory algorithm is x-optimal if the total number of time steps required for its execution T_{ext} satisfies $T_{ext} \leq x \cdot T_{seq}$. Here T_{seq} gives the minimum number of steps for solving the problem on a sequential computer with an infinite memory. It is said to be asymptotically x-optimal, if $T_{ext} \leq (x + o(1)) \cdot T_{seq}$. Here the hidden limit is to be taken for $M \to \infty$.*

Note that T_{seq} corresponds to the work that an algorithm has to do. In earlier papers [1, 4], the number of paging operations was considered as a unique quality measure for external-memory algorithms. In many cases this may be adequate, but generally, this is only half the story. There are external-memory problems, for which asymptotical one-optimality is achievable. Not considering the work of such an algorithm would amount to not considering its very essence.

Example 3 *Multiplication of two $n \times n$ matrices is a problem for which asymptotical one-optimality can be achieved (if we forget about the sub-cubic algorithms). As we have seen in Example 1, this problem can be solved such that each number is paged-in only $\mathcal{O}(n/\sqrt{M})$ times. Thus, the total paging takes $\mathcal{O}(n^3/\sqrt{M} \cdot G)$ time steps. It can be easily checked, that the number of computation steps equals that of the trivial row-times-column algorithm, which requires $\Omega(n^3)$ steps. Thus, the algorithm is $1 + \mathcal{O}(G/\sqrt{M})$ optimal. As G is a constant, this converges to 1 for M going to infinity.*

All this was known, but we had not seen before that the importance of also considering the work of an external-memory algorithm was so clearly exposed. One of the strong points of our BSP-like paradigm is that *both* important cost factors are 'automatically' taken into account. This gives more useful predictions of the performance.

In fact, we believe that the BSP-like paradigm is even more suited for the design and analysis of external-memory algorithms than the BSP model itself for parallel algorithms. We explain why. The BSP model is limited in three essential ways:

- The BSP model does not take the possibility of exploiting locality into account. On the communication time on a parallel computer, locality issues may be decisive. An attempt to cope with this limitation is made in [8].
- Start-up time, which makes sending small packets relatively expensive, is only very partially represented in the BSP model. An extension of the model taking such effects into account is given in [2].
- The computation is assumed to proceed in rounds.

For BSP-like algorithms running on a sequential computer, the first two points do not carry over. The hard-disc behaves like a completely connected network: the cost of a communication pattern is solely determined by the amount of data to be transferred plus the costs of a barrier synchronization. The third point remains. It is fundamental to any approach that calls itself BSP-like, and though this is certainly a limitation, we believe that operating in rounds is very natural, and essential for obtaining correct algorithms.

4.4 Typical Parameter Values

Values of M, N. Presently $M \simeq 10^7$, and $N \leq 10^{12}$.

Values of P, G, L. From the estimates for M and N, it follows that $P \leq 1000$. For a very fast computer, with a very slow hard-disc we may get $G = 100$, but mostly we will get $G < 10$. If there are several hard-discs, then G may even be around 1. L has a rather extreme value: $L \simeq M$, which is far larger than the value of l in most parallel computers. Thus, our system can be compared to a parallel system that communicates through powerful dial-up links.

Value of W_s/H_s. For hard problems with little 'locality', W_s/H_s is a constant: after a constant number of internal operations a new argument is necessary. This situation occurs for problems like list ranking.

In other problems, e.g., those that have good mesh algorithms, there is much less need for communication. More formally, algorithms for d-dimensional meshes, $d \geq 1$, a constant, can be simulated with $W_s/H_s = \Theta(M^{1/d})$. This implies that an x-work-optimal algorithm for a finite dimensional mesh leads to an asymptotically x optimal external-memory algorithm. In another guise, this idea has already been exploited in [16].

5 Designing BSP-Like Algorithms

One can try to design BSP-like algorithms from scratch. Obviously, minimization of the number of supersteps must be one of the principal goals, considering that L is much larger than G. Given this, the BSP-like paradigm provides a framework which allows for exact cost prediction. This is valuable in its own right, but in addition, we provide in this section an 'algorithms machine' for generating algorithms that work according to the paradigm. Only the latter gives our paradigm its full right of existence.

5.1 Inheritance of Quality

We restate the goal of external-memory computation to conform with our BSP-like framework:

Goal 1 *For a given problem of size N that must be solved on a computer with memory size M, the goal is to develop an algorithm that is x-optimal for the minimum x.*

Theorem 3 *Consider an external-memory problem of size N, with parameters (P, G, L). Suppose, that for parameters $(p, g, l) = (P, G, L)$ there is an x-optimal BSP-algorithm for solving this problem on a parallel computer. If no PU ever stores more than N/p data, then the corresponding BSP-like algorithm is also x-optimal.*

Proof: Let $T_{\text{par}}(p, g, l)$ be the time for a BSP algorithm, running on a parallel computer with parameters (p, g, l). For BSP algorithms, x-optimality is defined by

$$T_{\text{par}}(p, g, l) \leq x \cdot T_{\text{seq}}/p.$$

Multiplying the left and right sides with p, the theorem follows by combining with Theorem 2 and the definition of x-optimality for BSP-like algorithms. □

At this point we could conclude by supplying some more references to work on BSP algorithms: for many important problems extensive research has been performed on algorithms that give good, sometimes optimal, performance for large ranges of the parameters (p, g, l). All these results carry over immediately!

5.2 Limitations

Algorithms that Copy Data. In Theorem 2 and 3, we explicitly assume that in the BSP algorithm the PUs never store more than N/p data. This condition is necessary:

Example 4 *Consider computing $A \cdot B$, where A and B are $n \times n$ matrices. Initially each PU holds n^2/p entries of A and B. Now PU_i, $0 \leq i < p$, copies its entries of A to the PUs with indices $(i + k \cdot \sqrt{p}) \bmod p$, for all $0 \leq k < \sqrt{p}$; and all its B entries to the PUs with indices $(i + k) \bmod p$, $0 \leq k < \sqrt{p}$. Hereafter, all products can be computed without further communication. In order to compute the sums, one more round of communication is enough. The work in each PU is optimal, each PU exchanges at most $\mathcal{O}(n^2/\sqrt{p})$ data, and the number of routing rounds is two.*

If there is enough storage capacity in each PU, if l is very large, and if g is moderate, then the algorithm of the example may be competitive. However, the corresponding BSP-like algorithm does not make sense as an external-memory algorithm.

Geometrically Decreasing Problem Sizes. Theorem 3 holds generally, but there are good external-memory algorithms that are not found by looking for the best BSP algorithm with parameters (P, G, L). In this sense our approach shares the weaknesses of the PRAM approach (see Example 2). This is not due to the BSP-like paradigm itself, but rather to our definition of L.

The illustrated problem arises for algorithms with geometrically decreasing problem sizes. Generally, if during the algorithm the relevant set of data varies in size, then the number of 'PUs' can be varied accordingly. This is perfectly consistent with our BSP-like paradigm, and in (2), one only has to replace P by P_s, the number of PUs in Superstep. Such an algorithm can even be viewed as the sequential simulation of a BSP algorithm: it is easy to modify SEQUENTIAL_EXECUTION such that, if in a BSP algorithm the problem size in Superstep s is N_s, that the number of simulated PUs then equals $P_s = \lceil 3 \cdot N_s/M \rceil$.

The problem is best explained by an example:

Example 5 *Consider ranking a list of length $N > M$. Because in all algorithms, the work of a PU is only a small factor larger than the total amount it has to route, the* routing volume*, we may concentrate on the number of supersteps and the routing volume. What is the best BSP algorithm for $(p, g, l) = (P, G, L)$?*

We compare the two standard algorithms: pointer jumping and independent-set removal (better algorithms and details are provided in [15]). Pointer jumping can be performed in $\log N$ supersteps, with a volume of $2 \cdot \log N \cdot M$. Independent-set removal requires approximately $12 \cdot \log(N/M)$ supersteps, and routing volume $12 \cdot M$. So, the costs of these algorithms, $T_{par,\, poj}$ and $T_{par,\, isr}$, respectively, are given by

$$\begin{aligned} T_{par,\, poj} &\simeq 2 \cdot \log N \cdot (M/3) \cdot g + \log N \cdot l, \\ T_{par,\, isr} &\simeq 12 \cdot (M/3) \cdot g + 12 \cdot \log(N/M) \cdot l. \end{aligned}$$

Which algorithm is better depends on N and M, but especially on l/g. In our case, $l = (M/3) \cdot g$. For this l, pointer jumping is better iff

$$3 \cdot \log N < 12 + 12 \cdot \log(N/M).$$

For $N = 10^9$ and $M = 10^6$, this is the case. As a BSP algorithm, for these (p, g, l), pointer jumping will indeed be better than independent-set removal. However, the costs of the corresponding

BSP-like algorithms (neglecting the work) are given by

$$T_{\textit{ext, poj}} \simeq 3 \cdot \log N \cdot N \cdot G \simeq 90 \cdot N \cdot G,$$
$$T_{\textit{ext, isr}} \simeq \sum_s (3 \cdot (0.75^s \cdot N) \cdot G + 6 \cdot (0.75^s \cdot P) \cdot L) \leq 36 \cdot N \cdot G.$$

Thus, as an external-memory algorithm, independent-set-removal is clearly faster.

The problem is that Theorem 3 is not tight; it gives only a one-sided guarantee. The reason is that in a BSP algorithm there is real a difference between reducing the load in the PUs and reducing the number of PUs. In a BSP-like algorithm, if L is adapted, then this is more or less the same. For this fundamental reason we do not think that (P, G, L) can be defined such that Theorem 3 becomes tight: the correspondence between the problems is *not* one-to-one.

6 The Paradigm at Work

In the remainder of this paper we show how the BSP-like paradigm works for some basic problems. This is very small selection, but it demonstrates its potential.

Matrix Multiplication. We consider the multiplication of two $n \times n$ matrices on a sequential computer with $n < M < n^2$. In Example 3, we saw that the third algorithm from Example 1 is asymptotically one-optimal. Here we show that it is distinguished from the second algorithm in Example 1, by looking for the best BSP matrix-multiplication algorithm for the instance with $(p, g, l) = (3 \cdot n^2/M, G, M \cdot G/3)$.

For the work in the BSP algorithm, we only consider multiplications. Denote the time for the algorithms in Example 1 by $T_{\text{par, stripe}}$ and $T_{\text{par, square}}$, respectively. Then $T_{\text{par, square}}$ is clearly smaller:

$$\begin{aligned}
T_{\text{par, stripe}} &= \sum_{s=1}^{n^2 \cdot 3/M} (M^2/(9 \cdot n) + M/3 \cdot g + l) \\
&= n \cdot M/3 + n^2 \cdot g + n^2 \cdot 3/M \cdot l, \\
T_{\text{par, square}} &= \sum_{s=1}^{n \cdot \sqrt{3/M}} ((M/3)^{3/2} + M/3 \cdot g + l) \\
&= n \cdot M/3 + n \cdot \sqrt{M/3} \cdot g + n \cdot \sqrt{3/M} \cdot l.
\end{aligned}$$

Sorting. Next, we consider sorting N numbers. What is the best BSP algorithm for $(p, g, l) = (3 \cdot N/M, G, M \cdot G/3)$? We compare only the first and the last algorithm from Example 2: column-sort [10] and merge sort [6].

The time consumptions of these algorithms are denoted $T_{\text{par, column}}$ and $T_{\text{par, merge}}$, respectively. $T_{\text{seq, sort}}(N)$ is the time for sorting N numbers with the best sequential sorting algorithm. For $T_{\text{par, merge}}$ we give only a lower estimate. A modification of an algorithm in [14] gives a good BSP algorithm for column-sort consumption:

$$\begin{aligned}
T_{\text{par, column}} &= (1 + o(1)) \cdot T_{\text{seq, sort}}(M/3) + M \cdot g + 3 \cdot l = (1 + o(1)) \cdot T_{\text{seq}}(M/3), \\
T_{\text{par, merge}} &\geq T_{\text{seq, sort}}(M/3) + \Omega(M \cdot g) + \Omega(\log(M/N) \cdot l).
\end{aligned}$$

$T_{\text{par, merge}}$ is larger. Thus, again, the better external-memory algorithm can be found by just analyzing BSP performances.

Transitive Closure. Now we turn to problems for which so far we did not know an external-memory algorithm. The first problem is that of computing transitive closure of a general graph, and the related problems (LU decomposition, all pairs shortest paths, etc.), which sequentially are solved in $\mathcal{O}(n^3)$ time for a graph with n nodes.

For this problem, there is a BSP algorithm with the following running time [11]:

$$T_{\text{par, closure}} = n^3/p + n^2/\sqrt{p} \cdot g + \sqrt{p} \cdot l. \tag{3}$$

This requires only $\mathcal{O}(n^2/p)$ memory per PU, so the condition of Theorem 2 is satisfied:

Theorem 4 *With $N = n^2$, there is an external-memory transitive closure algorithm running in*

$$T_{\textit{ext, closure}}(N, M, G) = \mathcal{O}(N^{3/2} + N^{3/2} \cdot G/M^{1/2}).$$

Proof: Multiply (3) by p, and substitute $n = \sqrt{N}$, $P = \lceil 3 \cdot N/M \rceil$, $g = G$ and $L = M \cdot G/3$. The second and third term both contribute a term $\mathcal{O}(N^{3/2} \cdot G/M^{1/2})$. □

Fast Fourier Transform. We consider the fast Fourier transform, FFT, for a vector of length N. The sequential complexity of this problem is $\mathcal{O}(N \cdot \log N)$.

On a parallel computer with $p \leq \sqrt{N}$, there is a well-known BSP algorithm (a recent description is given in [17]):

$$T_{\text{par, FFT}} = \mathcal{O}(N \cdot \log N/p + N/p \cdot g + l).$$

Similarly to Theorem 4, this implies

Theorem 5 *If $M^2 \geq N$, then there is an external-memory algorithm for the FFT problem with running time $T_{\textit{ext, FFT}}$ as follows:*

$$T_{\textit{ext, FFT}}(N, M, G) = \mathcal{O}(N \cdot \log N + N \cdot G).$$

The results of Theorem 4 and Theorem 5 imply that for transitive closure and for FFT, the time for paging data is asymptotically negligible. In other words, both algorithms are asymptotically $\mathcal{O}(1)$-optimal.

7 Conclusion

We proposed the use of BSP-like algorithms as a general approach in the design and analysis of external-memory algorithms. With our paradigm, the work of an algorithm can be modeled in an integrated way. Earlier approaches mostly neglected the work, which is not correct in general. The quality of a BSP-like algorithm is expressed by its x-optimality. We provided a kind of machine for generating good external-memory algorithms, by proving an intimate link between BSP algorithms for parallel computers, and BSP-like algorithms: x-optimal BSP algorithms give rise to x-optimal BSP-like algorithms. In many, but not all, cases the optimal external-memory algorithm can be identified by searching for the optimal BSP algorithm. We demonstrated this for matrix-multiplication and sorting where the external-memory algorithms were already known. Furthermore we applied our paradigm to the known BSP algorithms for transitive closure and FFT and got (at least for us) new optimal external-memory algorithms back. Given the fact that there is far more research activity in the field of BSP algorithms (see [12] for references), it is to be expected that there are many more external-memory algorithms to be discovered in this way.

References

[1] Aggarwal, A., J.S. Vitter, 'The Input/Output Complexity of Sorting and Related Problems,' *Communications of the ACM*, 31(9), pp. 1116–1127, 1988.

[2] Bäumker, A. W. Dittrich, F. Meyer auf der Heide, 'Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP-Model,' *Proc. European Symposium on Algorithms*, LNCS 979, Springer-Verlag, pp. 17–30, 1995.

[3] Bilardi, G., F.P. Preparata, 'Horizons of Parallel Computation,' *Journal of Parallel and Distributed Computing*, 27, pp. 172–182, 1995.

[4] Chiang, Y-J, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter, 'External-Memory Graph Algorithms,' *Proc. 6th Symposium on Discrete Algorithms*, pp. 139–149, ACM-SIAM, 1995.

[5] Cormen, T.H., *Virtual Memory for Data Parallel Computing*, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[6] Cole, C., 'Parallel Merge Sort,' *SIAM Journal of Computing*, 17(4), pp. 770–785, 1988.

[7] JáJá, J., *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[8] Juurlink, B.H.H., H.A.G. Wijshoff, 'The E-BSP Model: Incorporating General Locality and Unbalanced Communication inti the BSP Model,' *Proc. 2nd International Euro-Par Conference*, LNCS 1124, Springer-Verlag, pp. 339–347, 1996.

[9] Kaufmann, M., J. F. Sibeyn, T. Suel, 'Derandomizing Algorithms for Routing and Sorting on Meshes,' *Proc. 5th Symp. on Discrete Algorithms*, pp. 669–679 ACM-SIAM, 1994.

[10] Leighton, F.T., 'Tight Bounds on the Complexity of Parallel Sorting,' *IEEE Transactions on Computers, C-34(4)*, pp. 344–354, 1985.

[11] McColl, W.F., 'Scalable Computing,' *Computer Science Today: Recent Trends and Developments*, J. van Leeuwen (Ed.), LNCS 1000, pp. 46–61, Springer-Verlag, 1995.

[12] McColl, W.F., 'Universal Computing,' *Proc. 2nd Euro-Par Conference*, LNCS 1123, pp. 25–36, Springer-Verlag, 1996.

[13] Patt, Y.N., 'The I/O Subsystem – A Candidate for Improvement,' *IEEE Computer*, 27(3), pp. 15–16, 1994.

[14] Sibeyn, J.F., 'Deterministic Routing and Sorting on Rings,' *Proc. 8th International Parallel Processing Symposium*, pp. 406–410, IEEE, 1994.

[15] Sibeyn, J.F., 'Better Trade-offs for Parallel List Ranking,' submitted to *Algorithmica*, 1996.

[16] Sibeyn, J.F., T. Harris, 'Exploiting Locality in LT-RAM Computation,' *Proc. 4th Scandinavian Workshop on Algorithm Theory*, LNCS 824, pp. 338–349, Springer-Verlag, 1994.

[17] Tishkin, A., 'The Bulk-Synchronous Parallel Random Access Machine,' *Proc. 2nd Euro-Par Conference*, LNCS 1124, pp. 327–338, Springer-Verlag, 1996.

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