Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 43–69. http://www.scpe.org
ISSN 1895-1767

EXTERNAL MEMORY IN BULK-SYNCHRONOUS PARALLEL ML*

FRÉDÉRIC GAVA†

Abstract. A functional data-parallel language called BSML was designed for programming Bulk-Synchronous Parallel algorithms, a model of computing which allows parallel programs to be ported to a wide range of architectures. BSML is based on an extension of the ML language with parallel operations on a parallel data structure called parallel vector. The execution time can be estimated. Dead-locks and indeterminism are avoided. For large scale applications where parallel processing is helpful and where the total amount of data often exceeds the total main memory available, parallel disk I/O becomes a necessity. In this paper, we present a library of I/O features for BSML and its formal semantics. A cost model is also given and some preliminary performance results are shown for a commodity cluster.

Key words. Parallel Functional Programming, Parallel I/O, Semantics, BSP.

1. Introduction. Some problems require performance that can only be provided by massively parallel computers. Programming these kind of computers is still difficult. Many important computational applications involve solving problems with very large data sets [44]. Such applications are also referred as *out-of-core* applications. For example astronomical simulation [47], crash test simulation [10], geographic information systems [32], weather prediction [52], computational biology [17], graphs [40] or computational geometry [11] and many other scientific problems can involve data sets that are too large to fit in the main memory and therefore fall into this category. For another example, the Large Hadron Collider of the CERN laboratory for finding traces of exotic fundamental particles (web page at `lhc-new-homepage.web.cern.ch`), when starts running, this instrument will produces about 10 Petabytes a month. The earth-simulator, the most powerful parallel machine in the top500 list, has 1 Petabyte of total main memory and 100 Petabytes of secondary memories. Using the main memory is not enough to store all the data of an experiment.

Using parallelism can reduce the computation time and increase the available memory size, but for challenging applications, the memory is always insufficient in size. For instance, in a mesh decomposition of a mechanical problem, a scientist might want to increase the mesh size. To increase the available memory size, a trivial solution is to use the *virtual memory* mechanism present in modern operating systems. This has been established as a standard method for managing external memory. Its main advantage is that it allows the application to access to a *large* virtual memory without having to deal with the intricacies of blocked secondary memory accesses. Unfortunately, this solution is inefficient if standard *paging policy* is employed [7]. To get the best performances, the algorithms must be restructured with explicit I/O calls on this secondary memory.

Such algorithms are generally called *external memory* (EM) algorithms and are designed for *large computational* problems in which the size of the internal memory of the computer is only a small fraction of the size of the problem ([55, 53] for a survey). Parallel processing is an important issue for EM algorithms for the same reasons that parallel processing is of practical interest in non-EM algorithm design. Existing algorithm and data structures were often unsuitable for out-of-core applications. This is largely due to the need of locality on data references, which is not generally present when algorithms are designed for internal memory due to the permissive nature of the PRAM model: parallel EM algorithms [54] are "new" and do not work optimally and correctly in "classical" parallel environments.

Declarative parallel languages are needed to simplify the programming of massively parallel architectures. Functional languages are often considered. The design of parallel programming languages is a tradeoff between the possibility to express the parallel features that are necessary for predictable efficiency (but with programs that are more difficult to write, prove and port) and the abstraction of such features that are necessary to make parallel programming easier (but which should not hinder efficiency and performance prediction). On the one hand the programs should be efficient but without the price of non portability and unpredictability of performances. The portability of code is needed to allow code reuse on a wide variety of architectures. The predictability of performances is needed to guarantee that the efficiency will always be achieved, whatever architecture is used.

*This work is supported by the ACI Grid program from the French Ministry of Research,under the project Caraml (`http://www.caraml.org`)

†Laboratory of Algorithms, Complexity and Logic (LACL), University of Paris XII, Val-de-Marne, 61 avenue du Général de Gaulle, 94010 Créteil cedex – France, `gava@univ-paris12.fr`

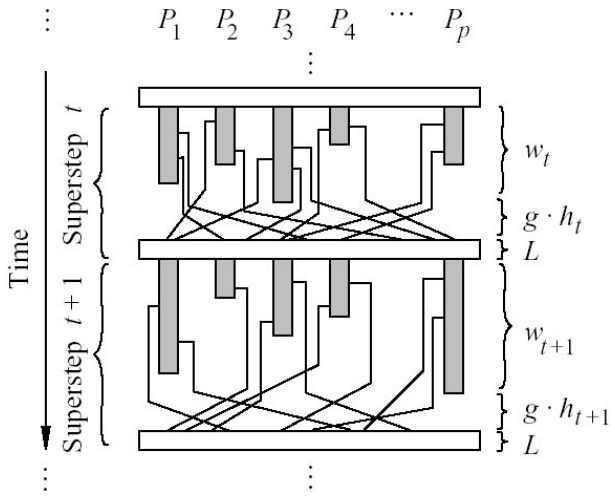


FIG. 2.1. *The BSP model of computation*

Another important characteristic of parallel programs is the complexity of their semantics. Deadlocks and non-determinism often hinder the practical use of parallelism by a large number of users. To avoid these undesirable properties, there is a trade-off between the expressiveness of the language and its structure which could decrease the expressiveness.

We are currently exploring the intermediate position of the paradigm of algorithmic skeletons [6, 42] in order to obtain universal parallel languages where the execution cost can easily be determined from the source code. In this context, cost means the estimate of parallel execution time. This last requirement forces the use of explicit processes corresponding to the processors of the parallel machine. Bulk-Synchronous Parallel ML or *BSML* is an extension of ML for programming Bulk-Synchronous Parallel algorithms as functional programs with a compositional cost model. Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [46, 50] to offer a high degree of abstraction like PRAM models and yet to allow portable and predictable performance on a wide variety of architectures with a realistic cost model based on a structured parallelism. Deadlocks and indeterminism are avoided. BSP programs are portable across many parallel architectures. Such algorithms offer predictable and scalable performances ([38] for a survey) and BSML expresses them with a small set of primitives taken from the *confluent* BSλ calculus [37]. Such operations are implemented as a library for the functional, with a strict evaluation strategy, programming language Objective Caml [33]. We refer to [27] for more details about the choice of this strategy for massively parallel computing.

Parallel disk I/O has been identified as a *critical component* of a suitable *high performance* computer. Research in EM algorithms has recently received considerable attention. Over the last few years, comprehensive computing and cost models that incorporate disks and multiple processors have been proposed [35, 55, 54], but not with all the above elements. [14, 16] showed how an EM machine can take full advantage of parallel disk I/O and multiple processors. This model is based on an extension of the BSP model for I/O accesses. Our research aims at combining the BSP model with functional programming. We naturally need to also extend BSML with I/O accesses for programming EM algorithms. This paper is the follow-up to our work on imperative features of our functional data-parallel language [22].

This paper describes a further step after [21] towards this direction. The remainder of this paper is organized as follows. First we review the BSP model in Section 2 and, then, briefly present the BSML language. In section 3 we introduce the EM-BSP model and the problems that appear in BSML. In section 4, we then give new primitives for our language. In section 5, we describe the formal semantics of our language with persistent features. Section 6 is devoted to the formal cost model associated to our language and Section 7 to some benchmarks of a parallel program. We discuss related work in section 8 and we end with conclusions and future research (section 9).

2. Functional Bulk-Synchronous Parallel ML.

2.1. Bulk-Synchronous Parallelism. A BSP computer contains a set of *processor-memory* pairs, a *communication network* allowing inter-processor delivery of messages and a *global synchronization unit* which

executes collective requests of a *synchronization barrier*. For the sake of conciseness, we refer to [5, 46] for more details. In this model, a parallel computation is subdivided into *supersteps* (Figure 2.1) at the end of which a barrier synchronization and a routing are performed. After that, all requests for data posted during a preceding superstep are fulfilled. The performance of the machine is characterized by 3 parameters expressed as multiples of the local processing speed r:

(i) p is the number of processor-memory pairs;
(ii) l is the time required for a global synchronization and
(iii) g is the time for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word. The network can deliver an h-relation in time $g \times h$ for any arity h.

These parameters can easily be obtained using benchmarks [28]. The execution time of a superstep s is thus the sum of the maximal local processing time, the maximal data delivery time and the global synchronization time, i.e, $\text{Time}(s) = \max_{i:processor} w_i^s + \max_{i:processor} h_i^s * g + l$ where $w_i^s =$ local processing time on processor i during superstep s and $h_i^s = \max\{h_{i+}^s, h_{i-}^s\}$ where h_{i+}^s (resp. h_{i-}^s) is the number of words transmitted (resp. received) by processor i during superstep s. The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of S supersteps is therefore the sum of 3 terms:

$$t_{comp} + t_{comm} + L \text{ where } \left\{ \begin{array}{rcl} t_{comp} & = & \sum_s \max_i w_i^s \\ t_{comm} & = & H \times g \text{ where } H = \sum_s \max_i h_i^s \\ L & = & S \times l. \end{array} \right.$$

In general t_{comp}, H and S are functions of p and of the size of data n, or of more complex parameters like data skew and histogram sizes. To minimize execution time, the BSP algorithm design must jointly minimize the number S of supersteps and the total volume h (resp. t_{comp}) and imbalance h^s (resp. t_{comm}) of communication (resp. local computation). Bulk Synchronous Parallelism and the Coarse-Grained Multicomputer (CGM), which can be seen as a special case of the BSP model are used for a large variety of applications. As stated in [13] "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures between the late eighties and the time from the mid-nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain".

bsp_p: unit→int **bsp_l**: unit→float **bsp_g**: unit→float
mkpar: (int→α)→α **par**
apply: ($\alpha \to \beta$)**par**→α **par**→β **par**
type α option = None | Some **of** α
put: (int→α option)**par**→(int→α option)**par**
at: α **par**→int→α

FIG. 2.2. *The Core Bsmllib Library*

2.2. Bulk-Synchronous Parallel ML. BSML does not rely on SPMD programming. Programs are usual "sequential" Objective Caml (OCaml) programs [33] but work on a parallel data structure. Some of the advantages are simpler semantics and better readability. The execution order follows the reading order in the source code (or, at least, the results are such as seems to follow the execution order). There is currently no implementation of a full BSML language but rather a partial implementation as a library for OCaml (web page at `http://bsmllib.free.fr/`).

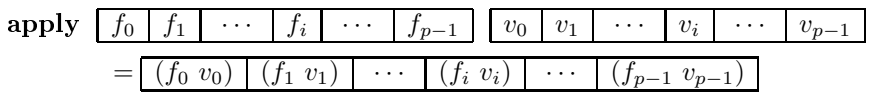
The so-called BSMLlib library is based on the elements given in Figure 2.2. They give access to the BSP parameters of the underling architecture: **bsp_p**() is p the *static* number of processes (this value does not change during execution), **bsp_g**() is g the time for collectively delivering a 1-relation and **bsp_l**() is l the time required for a global synchronization barrier.

There is an abstract polymorphic type α **par** which represents the type of p-wide parallel vectors of objects of type α one per processor. BSML parallel constructs operate on parallel vectors. Those parallel vectors are created by **mkpar** so that (**mkpar** f) stores (f i) on process i for i between 0 and $p-1$:

mkpar f = | (f 0) | (f 1) | ··· | (f i) | ··· | (f (p−1)) |

We usually write f as **fun** pid→e to show that the expression e may be different on each processor. This expression e is said to be *local*, i.e, a usual ML expression. The expression (**mkpar** f) is a parallel object and

it is said to be *global*. A usual ML expression which is not within a parallel vector is called *replicate*, i.e, identical to each processor. A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a superstep) and phases of global communication (second phase of a superstep) with global synchronization (third phase of a superstep). Asynchronous phases are programmed with **mkpar** and **apply** such that (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i) (e i)) on process i:



Let us consider the following expression:

let vf=**mkpar**(**fun** pid x→x+pid)
and vv=**mkpar**(**fun** pid→2∗pid+1)
in apply vf vv

The two parallel vectors are respectively equivalent to:

fun x→x + 0	**fun** x→x + 1	$\cdots$	**fun** x→x + i	$\cdots$	**fun** x→x + $(p-1)$

and

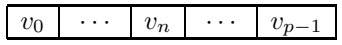
0	3	$\cdots$	$2\times i+1$	$\cdots$	$2\times(p-1)+1$

The expression **apply** vf vv is then evaluated to:

0	4	$\cdots$	$2\times i+2$	$\cdots$	$2\times(p-1)+2$

Readers familiar with BSPlib [28] will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by **put**. Consider the expression: **put**(**mkpar**(**fun** i→fs$_i$)) (∗). To send a value v from process j to process i, the function fs$_j$ at process j must be such that (fs$_j$ i) evaluates to Some v. To send no value from process j to process i, (fs$_j$ i) must evaluate to None. The expression (∗) evaluates to a parallel vector containing a function fd$_i$ of delivered messages on every process i. At process i, (fd$_i$ j) evaluates to None if process j sent no message to process i or evaluates to Some v if process j sent the value v to the process i.

The full language would also contain a synchronous projection operation **at**. (**at** vec n) returns the nth value of the parallel vector vec:



at expresses communication and synchronization phases. Without it, the global control cannot take into account data computed locally. Global conditional is necessary for expressing algorithms like: **Repeat** Parallel Iteration **Until** Max of local errors $< \epsilon$. The nesting of **par** types is prohibited and the projection should not be evaluated inside the scope of a **mkpar**. Our type system enforces these restrictions [23].

2.3. Examples.

2.3.1. Often Used Functions. Some useful functions can be defined by using only the primitives. For example the function replicate creates a parallel vector which contains the same value everywhere. The primitive **apply** can be used only for a parallel vector of functions which take only one argument. To deal with functions which take two arguments we need to define the apply2 function.

let replicate x = **mkpar**(**fun** pid→x)
let apply2 vf v1 v2 = **apply** (**apply** vf v1) v2

It is also common to apply the same sequential function at each process. This can be done using the parfun functions. They only differ in the number of arguments to apply:

let parfun f v = **apply**(replicate f) v
let parfun2 f v1 v2 = **apply**(parfun f v1) v2
let parfun3 f v1 v2 v3 = **apply**(parfun2 f v1 v2) v2

It is also common to apply a different function on a process. applyat n f_1 f_2 v applies function f_1 at process n and function f_2 at other processes:

```
let applyat n f1 f2 v =
 apply(mkpar(fun i→if i=n then f1 else f2)) v
```

2.3.2. Communication Function. Our example is the classical computation of the *prefix* of a list. Here we make the hypothesis that the elements of the list are distributed to all the processes as lists. Each processor performs a local reduction, then sends its partial result to the following processors and finally locally reduces its partial result with the sent values. Take for example the following expression:

scan_list_direct e (+) | [1;2] | [3;4] | [5] |

It will be evaluated to:

| $[e+1; e+1+2]$ | $[e+1+2+3; e+1+2+3+4;]$ | $[e+1+2+3+4+5]$ |

for a prefix of three processors and where e is the neutral element (here 0). To do this, we need first the computation of the prefix of a parallel vector:

```
(* scan_ direct:(α →α →α )→α →α par→α par *)
let scan_direct op e vv =
 let mkmsg pid v dst=if dst<pid then None else Some v in
 let procs_lists=mkpar(fun pid→from_to 0 pid) in
 let receivedmsgs=put(apply(mkpar mkmsg) vv) in
 let values_lists= parfun2 List.map
       (parfun (compose noSome) receivedmsgs) procs_lists in
  applyat 0 (fun _ →e) (List.fold_left op e) values_lists
```

where
$$\begin{cases} \text{List.map } f\ [v_0;\ldots;v_n] &= [(f\ v_0);\ldots;(f\ v_n)] \\ \text{List.fold\_left } f\ e\ [v_0;\ldots;v_n] &= f\ (\cdots(f\ (f\ e\ v_0)\ v_1)\cdots)\ v_n \\ \text{from\_to } n\ m &= [n; n+1; n+2;\ldots;m] \\ \text{noSome (Some } v) &= v \\ \text{compose } f\ g\ x &= (f\ (g\ x)). \end{cases}$$

Then, we can directly have the prefix of lists using some generic scan:

```
let scan_wide scan seq_scan_last map op e vv =
 let local_scan=parfun (seq_scan_last op e) vv in
 let last_elements=parfun fst local_scan in
 let values_to_add=(scan op e last_elements) in
 let pop=applyat 0 (fun x y→y) op in
  parfun2 map (pop values_to_add) (parfun snd local_scan)

let scan_wide_direct seq_scan_last map op e vv =
  scan_wide scan_direct seq_scan_last map op e vv

let scan_list scan op e vl =
 scan_wide scan seq_scan_last List.map op e vl
(* scan_list_direct:(α →α →α )→α →α list par→α list par *)
let scan_list_direct op e vl = scan_list scan_direct op e vl
```

where seq_scan_last $f\ e\ [v_0; v_1;\ldots; v_n] = (\text{last}, [(f\ e\ v_0); f(f\ e\ v_0)\ v_1;\ldots;\text{last}])$ where
last $= f\ (\cdots(f\ (f\ e\ v_0)\ v_1)\cdots)\ v_n$. The BSP cost formula of the above function (assuming op has a constant cost c_{op}) is thus $2 \times N \times c_{op} \times r\ + (p-1) \times s \times g\ +\ l$ where s denotes the size in words of a value compute by the scan and N the length of the biggest list held at a process. We have thus the time to compute the partial prefix, the time to send the partial results, time to perform the global synchronization and the time to finish the prefix.

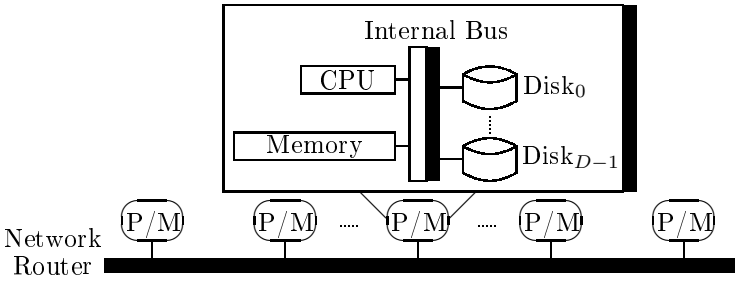


FIG. 3.1. *A BSP computer with external memories*

2.4. Advantages of Functional BSP Programming. One important benefit of the BSP model is the ability to accurately predict the execution time requirements of parallel algorithms. Communications are clearly separated from synchronization, i. e., this avoids deadlocks and it can be performed in any order, provided that the information is delivered at the beginning of the next superstep. This is achieved by constructing *analytical formulas* that are parameterized by a few values which captured the computation, communication and synchronization performance of the parallel system.

The clarity, abstraction and formal semantics of functional language make them desirable vehicles for complex software. The functional approach of this parallel model allows the re-use of suitable techniques from functional languages because a few number of parallel primitives is needed. Primitives of the BSML language with a strict strategy are derived from a confluent calculus [37] so parallel algorithms are also confluent and keep the advantages of the BSP models: no deadlock, efficient implementation using optimized communication algorithms, static cost formulas and cost previsions. The lazy evaluation strategy of pure functional language is not suited for the need of the massively parallel programmer. Lazy evaluation has the unwanted property of hiding complexity from the programmer [27]. The strict strategy of OCaml makes the BSMLlib a better tool for high performance applications because programs are transparent in the sense of making complexity explicit in the syntax.

Also, as in functional languages, we could easily *prove* and *certify* functional implementation of such algorithms with a proof assistant [1, 4] as in [20]. Using the *extraction* possibility of the proof assistant, we could generate a *certified* implementation to be used independently of the sequential or parallel implementation of the BSML primitives.

3. External Memory.

3.1. The EM-BSP model. Modern computers typically have several layers of memories which include the main memory and caches as well as disks. We restrict ourselves to the two-level model [54] because the speed difference between disks and the main memory is much more significant than between other layers of memories. [16] extended the BSP model to include secondary local memories. The basic idea is simple and it is illustrated in Figure 3.1. Each processor has, in addition to its local memory, an external memory (EM) in the form of a set of *disks*. This idea is applied to extend the BSP model to its EM version called **EM-BSP** by adding the following parameters to the standard BSP parameters:

(i) M is the local memory *size* of each processor;
(ii) D is the number of *disk drives* of each processor;
(iii) B is the *transfer block size* of a disk drive, and
(iv) G is the ratio of local computational capacity (number of local computation operations) divided by local I/O capacity (number of blocks of size B that can be transferred between the local disks and memory) per unit time.

In many practical cases, all processors have the *same number* of disks and, thus, the model is restricted to that case (although the model forbids different memory sizes). The disk drives of each processor are denoted by $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. Each drive consists of a sequence of *tracks* which can be accessed by direct random access. A track stores exactly one block of B words. Each processor can use all its D disk drives *concurrently* and transfer $D \times B$ words from/to the local disks to/from its local memory in a single I/O operation being at cost G. In such an operation, only one track per disk is permitted to be accessed *without any restriction* and each track is set on each disk. Note that an operation involving fewer disk drives incurs the same cost. Each processor is

assumed to be able to store in its local main memory at least some blocks from each disk at the same time, i. e., $M >> DB$.

Like computation on the BSP model, the computation of the EM-BSP model proceeds in a succession of supersteps. The communication costs are the same as for the BSP model. The EM-BSP model allows multiple I/O operations during the computation phase of the superstep. The total cost of each superstep is thus defined as $t_{comp,io} + t_{comm} + L$ where $t_{comp,io}$ is the computational cost and additional I/O cost charged for the supersteps, i.e, $t_{comp,io} = \sum_s \max_i(w_i^s + m_i^s)$ where m_i^s is the I/O cost incurred by processor i during superstep s. We refer to [16] to have the EM-BSP complexity of some classical BSP algorithms.

3.2. Examples of EM algorithms. Our first example is the matrix inversion which is used by many applications as a direct method to solve *linear systems*. The computation of the inverse of a matrix A can be derived from its LU factorization. [8] presents the LU factorization by blocks. For this parallel out-of-core factorization, the matrix is divided in blocks of columns called *superblocks*. The width of the superblock is determined by the amount of physical available memory: only blocks of the current superblock are in the main memory, the others are on disks. The algorithm factorize the matrix from left to right, superblock by superblock. Each time a new superblock of the matrix is fetched in the main memory (called the *active* superblock), all previous pivoting and update of a history of the right-looking algorithm are applied to the active superblocks. Once the last superblock is factorized, the matrix is re-read to apply the remaining row pivoting of the recursive phases. Note that the computation is done *data in place*, the matrix has been first distributed on processors and thus, for load balancing, a cyclic distribution of the data is used.

[9] presents PRAM algorithms using external-memory for graph problems as biconnected components of a graph or minimum spanning forest. One of them is the 3-coloring of a cycle applied to finding large independents sets for the problem of list ranking (determine, for each node v of a list, the rank of v define as the number of links from v to the end of the list). The methods for solving it is to update scattered successor and predecessor colors as needed after re-coloring a group of nodes of the list without sorting or scanning the entire list. As before, the algorithms works group by groups with only one group in the main memory.

The last example is the multi-string search problem which consists of determining which of k pattern strings occur in another string. Important applications on biological databases make use of very large text collections requiring specialized nontrivial search operations. [19] describes an algorithm for this problem with a constant number of supersteps and based on the distribution of a proper data structure among the processors and the disks to reduce and balance the communication cost. This data structure is based on a bind tree built on the suffixes of the strings and the algorithm works on longest common prefix on such trees and by lexicographic order. The algorithm takes advantage of disks by only keeping a part of a bind tree in the main memory and by collecting subpart of trees during the supersteps.

4. External Memory in BSML.

4.1. Problems by Adding I/O in BSML. The main problem by adding external memory and so I/O operators to BSML is to keep safe the fact that in the global context, the replicate values, i.e, usual OCaml values replicate on each processor, are the same. Such values are dedicated to the global control of the parallel algorithms. Take for example the following expression:

```
let chan=open_in "file.dat" in
 if (at (mkpar(fun pid→(pid mod 2)=0)) (input_value chan))
  then scan_direct (+) 0 (replicate 1)
 else (replicate 1)
```

It is not true that the file on each processor contains the same value. In this case, each processor reads on its secondary memory a different value. We would have obtained an incoherent result because each processor reads a different integer on the *channel* chan and some of them would execute scan_direct which need a synchronization. Others would execute replicate which does not need a synchronization. This breaks the confluent result of the BSML language and the BSP model of computation with its global synchronizations. If this expression had been evaluated with the BSMLlib library, we would have a breakdown of the BSP computer because **at** is a global synchronous primitive. Note that we also have this kind of problems in the BSPlib [28] where the authors note that only the I/O operations of the first processor are "safe". Another problem comes from *side-effects* that can occur on each processor. Take for example the following expression:

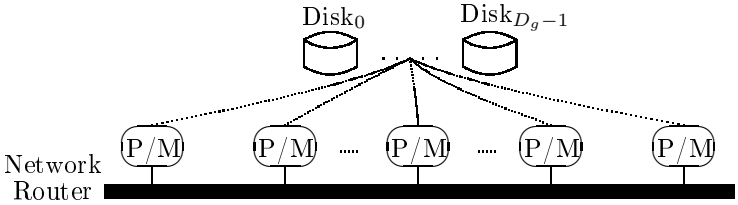


FIG. 4.1. *A BSP computer with shared disks*

```
let a=mkpar(fun i→if i=0 then(open_in "file.dat");()else ())
   in (open_out "file.dat")
```

where () is an empty value. If this expression had been evaluated with the BSMLlib library, only the first processor would have opened the file in a read mode. After, each processor opened the file with the same name in a write mode except the first one. This file has already been opened in read mode. We would also have an incoherent result because the first processor raised an exception which is not caught at all by other processes in the global context. This problem of side-effects could also be combined with the first problem if there is no file at the beginning of the computation. Take for example the following expression:

```
let chan=open_out "file.dat" in
let x=mkpar(fun i→if i=0 then (ouput_value 0) else ()) in
 ouput_value 1; close cha;
 let chan=open_in "file.dat" in
  if (at (mkpar(fun pid→(pid mod 2)=0)) (input_value chan))
   then scan_direct (+) 0 (replicate 1)
  else (replicate 1)
```

The first processor adds the integers 1 and 2 on its file and other processors add the integer 2 on their files. As in the first example, we would have a breakdown of the BSP computer because the integer read would not be the same and **at** is a global synchronous primitive.

4.2. The proposed solution. Our solution is to have two kinds of files: global and local ones. In this way, we have two kinds of I/O operators. Local I/O operators do not have to occur in the global context and global I/O ones do not have to occur locally. Local files are in local file systems which are presents in each processor as in the EM-BSP model. Global files are in a global file system. These files need to be the same from the point of view of each node. The global file system is thus in *shared disks* (as in Figure 4.1) or as a copy in each processor. They thus always give the same values for the global context. Note that if they are only shared disks and not local ones, the local file systems could be in different directories, one per processor in the global file system.

An advantage of having shared disks is the case of some algorithms which do not have distributed data at the beginning of the computation. As those which sort, the list of data to sort is in a global file at the beginning of the program and in another global file at the end. On the other hand, in the case of a distributed global file system, the global data are also distributed and programs are less sensitive to the problem of *faults*. Thus, we have two important cases for the global file system which could be seen as a new parameter of the EM-BSP machine: have we shared disks or not?

In the first case, the condition that the global files are the same for each processor point of view requires some synchronizations for some global I/O operators as created, opened or deleted a file. For example, it is impossible or un-deterministic for a processor to create a file in the global file system if at the same time another processor deleted it. On the other hand, reading (resp. writing) values from (resp. to) files do not need any synchronization. All the processors read the same values in the global file and only one of the processors needs to really write the value on the shared disks. In the case of a global output operator only one of the processors writes the value and in the case of a global input operator the value is first read from the disks by a processor and then is read by other processors from the operating system buffers. In this way, for all global operators, there is not a bottleneck of the shared disks.

In the second case, all the files, local and global ones, are distributed and no synchronization is needed at all. Each processor reads/writes/deletes etc. in its own file system. But at the beginning, the global file system needs to be empty or replicated to each processor and the global and local file systems in different directories.

Note that many modern parallel machines have concurrent shared disks. Such disks are always considered as *user disks*, i.e, disks where the users put the data needed for the computations whereas local disks are only generally used for the parallel computations of programs. For example, the earth simulator has 1,5 Petabytes for users as mass storage disks and a special network to access them. If there are no shared disks, NFS or scalable low level libraries as in [36] are able to simulate concurrent shared disks. Note also that if they are only shared disks, local disks could be simulated by using different directories for the local disks of the processors (one directory for one processor).

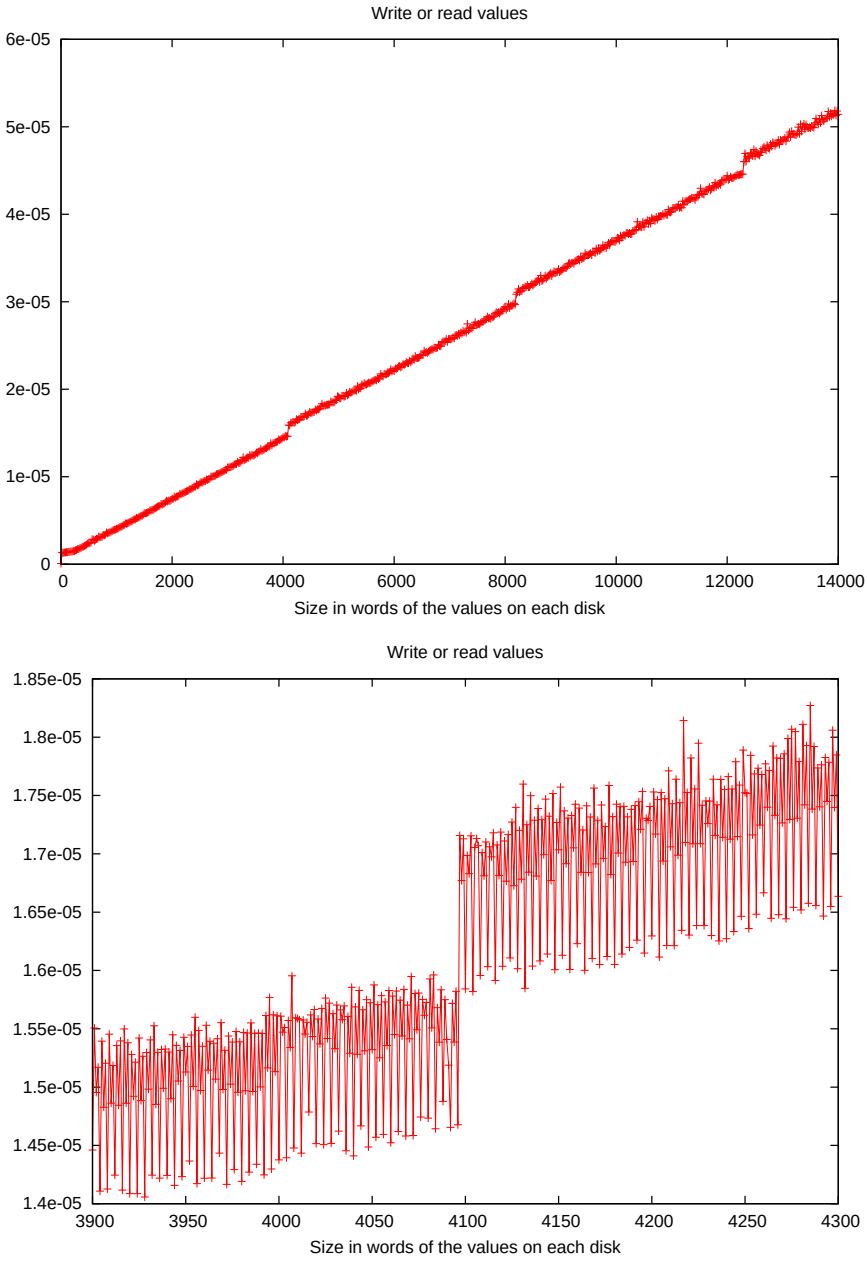


FIG. 4.2. *Benchmarks of EM parameters*

4.3. Our new model. After some experiments to determine the EM-BSP parameters of our parallel machine, we have found that operating systems do not read/write data in a constant time but in a linear time depending on the size of the data. We also notice that there is an overhead depending on the size of the blocks, i. e., if we have $n \times (DB) < s < (n+1) \times DB$, where s is the size in words of the data, there is $n+1$ overheads

to read/write this value from/to the D concurrent disks. Figure 4.2 gives the results of this experiment on a PC with 3 disks, each disk with blocks of 4096 words (seconds are plotted on the vertical axis). This program was run 10000 times and the average was taken. Such results are not altered if we decrease the number of disks.

Our proposed solution gives the processors access to two kinds of files: global and local ones. By this way, our model called **EM^2-BSP** extends the BSP model to its EM^2 version with two kinds of external memories, local and global ones. Each local file system will be on local concurrent disks as in the EM-BSP model. The global one will be on concurrent shared disks (as in Figure 4.1) if they exist or replicate on the local disks. The EM^2-BSP model is thus able to take into account the time to read the data and to distributed them into the processors. The following parameters are thus adding to the standard BSP parameters:

(i) M is the local memory size of each processor;
(ii) D^l is the number of independent disks of each processor;
(iii) B^l is the transfer block size of a local disk;
(iv) G^l is the time to read or write in parallel one word on each local disk;
(v) O^l is the overhead of the concurrent local disks;
(vi) D^g is the number of independent shared disks (or global disks);
(vii) B^g is the transfer block size of a global disk;
(viii) G^g is the time to read or write in parallel one word on each global disk and
(ix) O^g is the overhead of the concurrent global disks.

Of course, if there are no shared disks or no local disks: $D^l = D^g$, $B^l = B^g$, $G^l = G^g$ and $O^l = O^g$. A processor is able to read/write n words to its local disks in time $\lceil \frac{n}{D^l} \rceil \times G^l + \lceil \frac{n+1}{D^l B^l} \rceil \times O^l$ and n words to the global disks in time $\lceil \frac{n}{D^g} \rceil \times G^g + \lceil \frac{n+1}{D^g B^g} \rceil \times O^g$.

As in the EM-BSP model, the computation of the EM^2-BSP model proceeds in a succession of supersteps. The communication costs are the same as for the EM-BSP model and multiple I/O operations are also allowed during the computation phase of a superstep.

Note that G^g is not g even if processors access to the shared disks by the network (in case of some parallel machines): g is the time to perform a 1-relation and G^g is the time to read/write D words on the shared concurrent disks. It could depend on g in some parallel machine as clusters but it could depend on many other hardware parameters if, for example, there is a special network to access to the shared concurrent disks.

4.4. New Primitives. In this section we describe the core of our I/O library, i. e., the minimal set of primitives for programming EM^2-BSP algorithms. This library will be incorporated in the next release of the BSMLlib. This I/O library is based on the elements given in Figure 4.3. As in the BSMLlib library, we have functions to access to the EM^2-BSP parameters of the underlining architecture. For example, **embsp_loc_D**() is D^l the number of local disks and **glo_shared**() gives if the global file system is shared or not. Since we have two file systems, we need two kinds of names and two kinds of abstract types of output channels (resp. input channels): glo_out_channel (resp. glo_in_channel) and loc_out_channel (resp. loc_in_channel) to read/write values from/to global or local files.

We can open a named file for writing. The primitive returns a new output channel on that file. The file is truncated to zero length if it already exists. Either it is created or the primitive will raise an exception if the file could not be opened. For this, we have two kinds of functions for global and local files: (**glo_open_out** F) which opens the global file F in write mode and returns a global channel positioned at the beginning of that file and (**loc_open_out** f) which opens the local file f in write mode and returns a local channel positioned at the beginning of that file. In the same manner, we have two functions, **glo_open_in** and **loc_open_in** for opening a named file in read mode. Such functions return new local or global input channels positioned at the beginning of the files. In the case of global shared disks, a synchronization occurs for each global "**open**". With this global synchronization, each processor could signal to the other ones if it managed to open the file without errors or not and each processor would raise an exception if one of them has failed to open the file.

Now, with our channels, we can read/write values from/to the files. This feature is generally called *persistence*. To write the representation of a structured value of any type to a channel (global or local), we used the following functions: (**glo_output_value** Cha v) which writes the replicate value v to the opened global file and (**loc_output_value** cha v) which locally writes the local value v to the opened local file. The object can be then read back, by the reading functions: (**glo_input_value** Cha) (resp. (**loc_input_value** cha)) which returns from the global channel Cha (resp. local channel cha) the replicate value Some v (resp. local value) or None if there is no more value in the opened global file (resp. local file). This is the end of the file.

EM^2-BSP parameters

embsp_loc_D:unit→int	**embsp_loc_B**:unit→int	**embsp_loc_G**:unit→float
embsp_glo_D:unit→int	**embsp_glo_B**:unit→int	**embsp_glo_G**:unit→float
embsp_loc_O:unit→float	**embsp_glo_O**:unit→float	**glo_shared**:unit→bool

Global I/O primitives	Local I/O primitives
glo_open_out:glo_name→glo_out_channel	**loc_open_out**: loc_name→loc_out_channel
glo_open_in:glo_name→glo_in_channel	**loc_open_in**:loc_name→loc_out_channel
glo_output_value:glo_out_channel→α→unit	**loc_output_value**:loc_out_channel→α→unit
glo_input_value:glo_in_channel→α option	**loc_input_value**:loc_in_channel→α option
glo_close_out:glo_out_channel→unit	**loc_close_out**:loc_out_channel→unit
glo_close_in:glo_in_channel→unit	**loc_close_in**:loc_in_channel→unit
glo_delete:glo_name→unit	**loc_delete**:loc_name→unit
glo_seek:glo_in_channel→int→unit	**loc_seek**:loc_in_channel→int→unit

From local to global

glo_copy:int→loc_name→glo_name→unit

FIG. 4.3. *The Core I/O Bsmllib Library*

Such functions read the representation of a structured value and we refer to [34] about having type safety in channels and reading them in a safe way. We also have (**glo_seek** Cha n) (resp. **loc_seek**) which allows to positioned the channel at the nth value of a global file (resp. local file). The behavior is unspecified if any of the above functions is called with a closed channel.

Note that only local or replicate values could be written on local or global files. Nesting of parallel vectors is prohibited and thus **loc_output_value** could only write local values. It is also impossible to write on a shared global file a parallel vector of values (global values) because these values are different on each processor and **glo_output_value** is an asynchronous primitive. Such values could be written in any order and could be mixed with other values. This is why only local and replicate values should be read/write from/to disks (see section 6 for more details).

After, read/write values from/to channels, we need to close them. As previously, we need four kinds of functions: two for the input channels (local and global ones) and two for the output channels. For example, (**glo_close_out** Cha), closes the global output channel Cha which had been created by **glo_open_out**. The **glo_delete** and **loc_delete** primitives delete a global or a local file if it is first closed.

The last primitive copies a local file from a processor to the global file system. It is thus a global primitive. (**glo_copy** n f F) copies the file f from the processor n to the global file system with the name F. This primitive could be used at the end of a BSML program to copy the local results from local files to the global (user) file system. It is not a communication primitive because used as a communication primitive, **glo_copy** has a more expensive cost than any communication primitive (see section 6). In the case of a distributed global file system, the file is duplicated on all the global file systems of each processor and thus all the data of the file are all put into the network. On the contrary, in the case of global shared disks, it is just a copy of the file because, access to the global shared disks is generally slower than putting values into the network and read them back by another processor.

Using these primitives, the final result of any program would be the same (but naturally without the same total time, i. e., without the same costs) with shared disk or not. Now, to better understand how these new primitives work, we describe a formal semantics of our language with such persistent features.

5. High Order Formal Semantics.

5.1. Mini-BSML. Reasoning on the complete definition of a functional and parallel language such as BSML, would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this section introduces a core language as a mini programming language. It is an attempt to trade between integrating the principal features of persistence, functional, BSP language and being simple. The

expressions of mini-BSML, written e possibly with a prime or subscript, have the following abstract syntax:

e ::=	x	variables	$\mid$ c	constants
$\mid$	**op**	operators	$\mid$ **fun** $x \rightarrow e$	abstraction
$\mid$	$(e\ e)$	application	$\mid$ **let** $x = e$ **in** e	binding
$\mid$	(e, e)	pairs	$\mid$ **if** e **then** e **else** e	conditional
$\mid$	(**mkpar** e)	parallel vector	$\mid$ (**apply** $e\ e$)	parallel application
$\mid$	(**put** e)	communication	$\mid$ (**at** $e\ e$)	projection
$\mid$	f	file names or channels		

In this grammar, x ranges over a countable set of identifiers. The form $(e\ e')$ stands for the application of a function or an operator e, to an argument e'. The form **fun** $x \rightarrow e$ is the so-called and well-known lambda-abstraction that defines the first-class function of which the parameter is x and the result is the value of e. Constants c are the integers, the booleans, the number of processes **p** and we assume having a unique value for the type `unit`: **()**. The set of primitive operations *op* contains arithmetic operations, pair operators, test function **isnc** of the **nc** constructor which plays the role of the None constructor in OCaml, fixpoint to defined natural iteration functions and our I/O operators: $\mathbf{open^r}$ (resp. $\mathbf{open^w}$) to open a file as a channel in read mode (resp. write mode), $\mathbf{close^r}$ (resp. $\mathbf{close^w}$) to close a channel in read mode (resp. write mode), **read**, **write** to read or write in a channel, **delete** to delete a file and **seek** to change the reading position. All those operators are distinguished with a subscript which is l for a local operator and g for a global one. We also have our parallel operators: **mkpar**, **apply**, **put** and **at**. We also have two kinds of file systems, the local and the global ones, defined with (possibly with a prime):

- f for a file name;
- $f_{\mathbf{w}}$ for a write channel, $f_{\mathbf{r}}$ for a read channel and g_k^{ξ} for a channel pointed on the kth value of a file where ξ is the name of the channel;
- ${}^{?}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right]$ is a file where ? is **c**, **r** or **w** for a close file or an open file in read or write mode and where $v_0, \ldots, v_n$ the values hold in the file.

When a file is opened in read mode, it contains the name $[g_n^a, \ldots, g_m^z]$ of the channels that pointed to it and the position of these channels. Before presenting the dynamic semantics of the language, i. e., how the expressions of mini-BSML are computed to *values*, we present the values themselves and the simple ML types [39] of the values. There is one semantics per value of p, the number of processes of the parallel machine. In the following, the expressions are extended with the parallel vectors: $\langle e, \ldots, e \rangle$ (nesting of parallel vectors is prohibited; our static analysis enforces this restriction [23]). The values of mini-BSML are defined by the following grammar:

v ::=	**fun** $x\ \rightarrow e$	functional value	$\mid$ c	constant
	$\mid$ **op**	primitive	$\mid$ (v, v)	pair value
	$\mid$ $\langle v, \ldots, v \rangle$	p-wide parallel vector value	$\mid$ f	file names or channels

and the simple ML types of values are defined by the following grammar:

τ ::=	κ	base type (bool, int, unit, file names or channels)	$\mid$ α	type variables
	$\mid$ $\tau_1 \rightarrow \tau_2$	type of functional values from τ_1 to τ_2	$\mid$ $\tau_1 * \tau_2$	type for pair values

We note $\vdash v : \tau$ to say that the value v has the type τ and we refer to [39] for an introduction to the types of the ML language and to [23] for those of BSML.

5.2. High Order Semantics. The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we used a small-step semantics. It consists of a predicate between an expression and another expression defined by a set of axioms and rules called steps. The small-step semantics describes all the steps of the language from an expression to a value. We suppose that we evaluate only expressions that have been type-checked [23] (nesting of parallel vectors has been prohibited). Unlike in a sequential computer with a sequential programming language, a unique file system (a set of files) for persistent operators is not sufficient. We need to express the file system of all our processors and our global file system. We assume a finite set $\mathcal{N} = \{0, \ldots, p-1\}$ which represents the set of processor names and we write i for these names and $\bowtie$ for the whole parallel computer. Now, we can formalize the files for each processor and for the network. We write $\{f_i\}$ for the file system of processor i with $i \in \mathcal{N}$. We assume that each processor has a file system as an infinite mapping of files which are different at each processor. We write $\{f\} = \{\{f_0\}, \ldots, \{f_{p-1}\}\}$ for all the local file systems of our parallel machine and $\{\mathcal{F}\}$ for our global file

$$
\begin{array}{llll|llll}
(\mathbf{bsp\_p}\ ()) & \xrightarrow[\delta]{} & \mathbf{p} & & (+\ (n_1, n_2)) & \xrightarrow[\delta]{} & n & \text{with } n = n_1 + n_2 \\
(\mathbf{fst}\ (v_1, v_2)) & \xrightarrow[\delta]{} & v_1 & & (\mathbf{snd}\ (v_1, v_2)) & \xrightarrow[\delta]{} & v_2 & \\
\mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2 & \xrightarrow[\delta]{} & e_1 & & \mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2 & \xrightarrow[\delta]{} & e_2 & \\
(\mathbf{isnc\ nc}) & \xrightarrow[\delta]{} & \mathbf{true} & & (\mathbf{isnc}\ v) & \xrightarrow[\delta]{} & \mathbf{false} & \text{if } v \neq \mathbf{nc} \\
(\mathbf{fix\ op}) & \xrightarrow[\delta]{} & \mathbf{op} & & (\mathbf{fix}\ (\mathbf{fun}\ x \to e)) & \xrightarrow[\delta]{} & e[x \leftarrow (\mathbf{fix}\ (\mathbf{fun}\ x \to e))] &
\end{array}
$$

FIG. 5.1. *Functional δ-rules*

system. The persistent version of the small-steps semantics has the following form: $\{\mathcal{F}\}/e/\{f\} \rightharpoonup \{\mathcal{F}'\}/e'/\{f'\}$. We note $\stackrel{*}{\rightharpoonup}$, for the transitive and reflexive closure of $\rightharpoonup$, e. g., we note $\{\mathcal{F}^0\}/e_0/\{f^0\} \stackrel{*}{\rightharpoonup} \{\mathcal{F}\}/v/\{f\}$ for $\{\mathcal{F}^0\}/e_0/\{f^0\} \rightharpoonup \{\mathcal{F}^1\}/e_1/\{f^1\} \rightharpoonup \{\mathcal{F}^2\}/e_2/\{f^2\} \rightharpoonup \ldots \rightharpoonup \{\mathcal{F}\}/v/\{f\}$. To define the relation $\rightharpoonup$, we begin with some rules for two kinds of reductions:

(i) $e/\{f_i\} \stackrel{i}{\rightharpoonup} e'/\{f_i'\}$ which could be read as "with the initial local file system $\{f_i\}$, at processor i, the expression e is reduced to e' with the file system $\{f_i'\}$";

(ii) $\{\mathcal{F}\}/e/\{f\} \stackrel{\bowtie}{\rightharpoonup} \{\mathcal{F}'\}/e'/\{f\}$ which could be read as "with the initial global file system $\{\mathcal{F}\}$ and with the initial set of local file systems, the expression e is reduced to e' with the global file system $\mathcal{F}'$ and with the same set of local file systems".

To define these relations, we begin with some axioms for the functional head reduction $\stackrel{\varepsilon}{\rightharpoonup}$:

$$(\mathbf{fun}\ x \to e)\ v \quad \stackrel{\varepsilon}{\rightharpoonup} \quad e[x \leftarrow v] \qquad \text{and} \qquad \mathbf{let}\ x = v\ \mathbf{in}\ e \quad \stackrel{\varepsilon}{\rightharpoonup} \quad e[x \leftarrow v]$$

We write $e[x \leftarrow v]$ for the expression obtained by substituting all the free occurrences of x in e by v. Free occurrences of a variable are defined as a classical and trivial inductive function on our expressions. This functional head reduction has two versions. First, a local reduction, $\stackrel{\varepsilon}{\underset{i}{\rightharpoonup}}$, of just the processor i and second, a global reduction, $\stackrel{\varepsilon}{\underset{\bowtie}{\rightharpoonup}}$, of the whole parallel machine:

$$\frac{e \stackrel{\varepsilon}{\rightharpoonup} e'}{e\ /\ \{f_i\} \stackrel{\varepsilon}{\underset{i}{\rightharpoonup}} e'\ /\ \{f_i\}}\ (1) \qquad \frac{e \stackrel{\varepsilon}{\rightharpoonup} e'}{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \stackrel{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} \{\mathcal{F}\}\ /\ e'\ /\ \{f\}}\ (2)$$

For primitive operators we also have some axioms, the δ-rules. The functional δ-rules $\underset{\delta}{\rightharpoonup}$ are given in Figure 5.1. First, we have functional δ-rules which could be used by one processor i, $\underset{\delta_i}{\rightharpoonup}$ or by the parallel machine, $\underset{\delta_\bowtie}{\rightharpoonup}$. As in the functional head reduction, we have two different cases for using functional δ-rules:

$$\frac{e \underset{\delta}{\rightharpoonup} e'}{e\ /\ \{f_i\} \underset{\delta_i}{\rightharpoonup} e'\ /\ \{f_i\}}\ (3) \qquad \frac{e \underset{\delta}{\rightharpoonup} e'}{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \underset{\delta_\bowtie}{\rightharpoonup} \{\mathcal{F}\}\ /\ e'\ /\ \{f\}}\ (4)$$

Such reductions, which are not persistent reductions, do not change and do not need the files. Only persistent operators change and need them.

$$
\begin{array}{l}
\{\mathcal{F}\}\ /\ (\mathbf{mkpar}\ v)\ /\ \{f\} \quad \underset{\delta_\diamond}{\rightharpoonup} \quad \{\mathcal{F}\}\ /\ \langle (v\ 0), \ldots, (v\ (p-1)) \rangle\ /\ \{f\} \\
\{\mathcal{F}\}/(\mathbf{apply}\ \langle v_0, \ldots, v_{p-1} \rangle\ \langle v_0', \ldots, v_{p-1}' \rangle)\ /\ \{f\} \quad \underset{\delta_\diamond}{\rightharpoonup} \quad \{\mathcal{F}\}/\langle (v_0\ v_0'), \ldots, (v_{p-1}\ v_{p-1}') \rangle/\{f\} \\
\{\mathcal{F}\}\ /\ (\mathbf{at}\ \langle \ldots, v_n, \ldots \rangle\ n)\ /\ \{f\} \quad \underset{\delta_\diamond}{\rightharpoonup} \quad \{\mathcal{F}\}\ /\ v_n\ /\ \{f\} \ \text{ if } \mathcal{A}_c(v_n) \neq \mathbf{True} \\
\{\mathcal{F}\}\ /\ (\mathbf{put}\ \langle v_0, \ldots, v_{p-1} \rangle)\ /\ \{f\} \underset{\delta_\diamond}{\rightharpoonup} \{\mathcal{F}\}\ /\ (\mathbf{mkfun}\ (\langle \mathbf{send}\ (\mathbf{init}\ v_0\ \mathbf{p}), \ldots, \mathbf{send}\ (\mathbf{init}\ v_{p-1}\ \mathbf{p}) \rangle))\ /\ \{f\} \\
\{\mathcal{F}\}\ /\ \langle \mathbf{send}\ [v_0^0, .., v_0^{p-1}], \ldots, \mathbf{send}\ [v_{p-1}^0, .., v_{p-1}^{p-1}] \rangle\ /\ \{f\} \\
\qquad \underset{\delta_\diamond}{\rightharpoonup} \{\mathcal{F}\}\ /\ \langle [v_0^0, .., v_{p-1}^0], \ldots, [v_0^{p-1}, .., v_{p-1}^{p-1}] \rangle\ /\ \{f\} \text{ if } \forall n, m \in 0, \ldots, p-1\ \mathcal{A}_c(v_n^m) \neq \mathbf{True}
\end{array}
$$

where $\mathbf{mkfun} = \mathbf{apply}\ (\mathbf{mkpar}\ (\mathbf{fun}\ j\ t\ i \to \mathbf{if}\ (\mathbf{and}\ (\leq (0, i), < (i, \mathbf{p})))\ \mathbf{then}\ (\mathbf{access}\ t\ i)\ \mathbf{else}\ \mathbf{nc}))$

FIG. 5.2. *Parallel δ-rules*

Second, for the parallel primitives, we naturally have δ-rules but we do not have those δ-rules on a single processor but only for the parallel machine (Figure 5.2). For simple reasons it is impossible for a processor to send a channel to another processor. This second processor does not have to read in this channel because it could be seen as a hidden communication. In this way, we have to test if the sent values contain channels or not. To do this, we used a trivial inductive function $\mathcal{A}_c$ which tells whether an expression contains a channel or not. Note that this work is done when OCaml serializes values. This raises an exception when an abstract datum like a channel has been found. The evaluation of a **put** primitive proceeds in two steps. In a first step, each processor creates a *pure functional array* of values. Thus, we need a new kind of expression, arrays written $[e, \ldots, e]$. **init** and **access** operators are used to manipulate these functional arrays:

$$\textbf{access}\ [v_0, \ldots, v_n, \ldots, v_m]\ n \ \xrightarrow[\delta]{}\ v_n \qquad \text{and} \qquad \textbf{init}\ f\ m \ \xrightarrow[\delta]{}\ [(f\ 0), \ldots, (f\ (m\text{–}1))]$$

In the second step, the **send** operations exchange these arrays. For example, the value at the index j of the array held at process i is sent to process j and is stored at index i of the result. The function **mkfun** constructs a parallel vector of functions from the resulting vector of arrays.

$$(\textbf{open}^{\mathbf{r}}\ f)/\{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\} \qquad \xrightarrow[\delta]{io} \quad f^a_{\mathbf{r}}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a_0], \ldots, f''\}$$

$$(\textbf{open}^{\mathbf{r}}\ f)/\{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, \ldots, g^z], \ldots, f''\} \xrightarrow[\delta]{io} f^\xi_{\mathbf{r}}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, \ldots, g^z, g^\xi_0], \ldots, f''\}$$

$$(\textbf{open}^{\mathbf{w}}\ f)/\{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\} \qquad \xrightarrow[\delta]{io} \quad f^\xi_{\mathbf{w}}\ /\ \{f', \ldots, {}^{\mathbf{w}}f\left[\begin{smallmatrix} \emptyset \end{smallmatrix}\right], \ldots, f''\}$$

$$(\textbf{open}^{\mathbf{w}}_l\ f)/\{f', \ldots, f''\} \quad \xrightarrow[\delta]{io} \quad f^\xi_{\mathbf{w}}\ /\ \{f', \ldots, {}^{\mathbf{w}}f\left[\begin{smallmatrix} \emptyset \end{smallmatrix}\right], \ldots, f''\} \text{ if } f \notin \{f', \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, \ldots, g^\xi_k, \ldots, g^z], \ldots, f''\} \quad \xrightarrow[\delta]{io} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, \ldots, g^z], \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, \ldots, g^z], \ldots, f''\} \qquad \xrightarrow[\delta]{io} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, \ldots, g^z], \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^\xi_k], \ldots, f''\} \qquad \xrightarrow[\delta]{io} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\} \qquad \xrightarrow[\delta]{io} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{w}}\ f^\xi_{\mathbf{w}}/\{f', \ldots, {}^{?}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\} \quad \xrightarrow[\delta]{io} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\} \text{ where } ? = \mathbf{w} \text{ or } ? = \mathbf{c}$$

$$(\textbf{read}^\tau\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} \vdots \\ v_k \\ \vdots \end{smallmatrix}\right][g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\} \xrightarrow[\delta]{io} v_k\ /\ \{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} \vdots \\ v_k \\ \vdots \end{smallmatrix}\right][g^a, .., g^\xi_m, \ldots, g^z], \ldots, f''\}$$
$$\text{if} \vdash v_k : \tau \text{ and } m = k + 1.\ v_k \text{ is the } k\text{th value of } f$$

$$(\textbf{read}^\tau\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\} \xrightarrow[\delta]{io} \mathbf{nc}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\}$$
$$\text{if } k > n$$

$$(\textbf{seek}\ f^\xi_{\mathbf{r}}\ k)/\{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, .., g^\xi_m, \ldots, g^z], \ldots, f''\} \xrightarrow[\delta]{io} \mathbf{nc}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right][g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\}$$

$$(\textbf{write}\ (v, f^\xi_{\mathbf{w}}))/\{f', \ldots, {}^{\mathbf{w}}f\left[\begin{smallmatrix} \vdots \end{smallmatrix}\right], \ldots, f''\} \xrightarrow[\delta]{io} ()/\{f', \ldots, {}^{\mathbf{w}}f\left[\begin{smallmatrix} v \\ \vdots \end{smallmatrix}\right], \ldots, f''\} \text{ if } \mathcal{A}_c(v_n) \neq \textbf{True}$$

$$(\textbf{delete}\ f)/\{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} \vdots \end{smallmatrix}\right], \ldots, f''\} \qquad \xrightarrow[\delta]{io} \quad ()/\{f', \ldots, f''\}$$

FIG. 5.3. *δ-rules of the persistent operators*

Third, we complete our semantics by giving the δ-rules $\xrightarrow[\delta]{io}$ of the I/O operators given in Figure 5.3. The **open** operation opens a file (in read or write mode) and returns a new channel, pointing to this file, to access to the values of the file or write values in this file. Opening a file in write mode, gives an empty file. If possible, $\textbf{read}^\tau$ gives the value of type τ contained in the file from the channel. If no more value could be read then $\textbf{read}^\tau$ returns an empty value. The **write** operation writes a new value into the file using the channel. **delete**

$$
\begin{array}{rcl}
\Gamma & ::= & [] \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & \textbf{let}\ x = \Gamma\ \textbf{in}\ e \\
& | & (\Gamma, e) \\
& | & (v, \Gamma) \\
& | & \textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Gamma) \\
& | & (\textbf{apply}\ \Gamma\ e) \\
& | & (\textbf{apply}\ v\ \Gamma) \\
& | & (\textbf{put}\ \Gamma) \\
& | & (\textbf{at}\ \Gamma\ e) \\
& | & (\textbf{at}\ v\ \Gamma)
\end{array}
\qquad
\begin{array}{rcl}
\Gamma_l^i & ::= & \Gamma_l^i\ e \\
& | & v\ \Gamma_l^i \\
& | & \textbf{let}\ x = \Gamma_l^i\ \textbf{in}\ e \\
& | & (\Gamma_l^i, e) \\
& | & (v, \Gamma_l^i) \\
& | & \textbf{if}\ \Gamma_l^i\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Gamma_l^i) \\
& | & (\textbf{apply}\ \Gamma_l^i\ e) \\
& | & (\textbf{apply}\ v\ \Gamma_l^i) \\
& | & (\textbf{put}\ \Gamma_l^i) \\
& | & (\textbf{at}\ \Gamma_l^i\ e) \\
& | & (\textbf{at}\ v\ \Gamma_l^i) \\
& | & \langle e, \ldots, \overbrace{\Gamma_l}^{i}, e, \ldots, e \rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma_l & ::= & [] \\
& | & \Gamma_l\ e \\
& | & v\ \Gamma_l \\
& | & \textbf{let}\ x = \Gamma_l\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma_l\ \textbf{in}\ e \\
& | & (\Gamma_l, e) \\
& | & (v, \Gamma_l) \\
& | & \textbf{if}\ \Gamma_l\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{send}\ \Gamma_l) \\
& | & [\Gamma_l, e_1, \ldots, e_n] \\
& | & [v_0, \Gamma_l, \ldots, e_n] \\
& | & \ldots \\
& | & [v_0, v_1, \ldots, \Gamma_l]
\end{array}
$$

FIG. 5.4. *Context of evaluation*

deletes a file from the file system if it has been fully closed. **close** closes a channel or do nothing if the channel has been first closed. All those rules could be distinguished with a subscript (l or g) for the local or the global operators. Thus, we need two kinds of reductions, one for the local reduction $\xrightarrow[\delta_i]{io}$ and another one for the global reduction $\xrightarrow[\delta_{\bowtie}]{io}$:

$$\frac{e\ /\ \{f_i\} \xrightarrow[\delta]{io} e'\ /\ \{f_i'\}}{e\ /\ \{f_i\} \xrightarrow[\delta_i]{io} e'\ /\ \{f_i'\}}\ (5) \qquad \frac{e\ /\ \{\mathcal{F}\} \xrightarrow[\delta]{io} e'\ /\ \{\mathcal{F}'\}}{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \xrightarrow[\delta_{\bowtie}]{io} \{\mathcal{F}'\}\ /\ e'\ /\ \{f\}}\ (6)$$

First, for a single processor i such persistent operations work on the local file system of the processor i where they are executed. Second, for the whole parallel machine, we have the same operations except for the global file system. The special operator $\mathbf{copy}_{\bowtie}$ copies one file of one processor into the global file system:

$$\{F', .., F''\}/(\mathbf{copy}\ i\ f\ \mathcal{F})/\{f_0, \ldots, f_i, \ldots, f_{p-1}\} \xrightarrow[\delta_{\bowtie}]{io} \{F', .., F'', {}^{\mathbf{c}}F\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right]\}/()/\{f_0, \ldots, f_i, \ldots, f_{p-1}\}$$
$$\text{if } F \notin \{F', .., F''\} \text{ and } f_i = \{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\}$$

Now, the complete definitions of our two kinds of reductions are:

$$\xrightarrow{i} \ = \ \xrightarrow[i]{\varepsilon} \cup \xrightarrow[\delta_i]{} \cup \xrightarrow[\delta_i]{io} \qquad \text{and} \qquad \xrightarrow{\bowtie} \ = \ \xrightarrow[\bowtie]{\varepsilon} \cup \xrightarrow[\delta_{\bowtie}]{} \cup \xrightarrow[\delta_{\not\bowtie}]{} \cup \xrightarrow[\delta_{\bowtie}]{io}$$

5.3. Contexts of evaluation. It is easy to see that we cannot always make a head reduction. We have to reduce "in depth" in the sub-expressions. To define this deep reduction, we define two kinds of contexts, i.e, expressions with a *hole* noted [] that have the abstract syntax given in Figure 5.4. The hole gives where expressions could be reduced. In this way, the contexts give the order of evaluation of the arguments of the construction of the language, i.e, the strategy of the language.

The Γ context is used to define a global reduction of the parallel machine. For example:

$$\Gamma = \textbf{let}\ x = []\ \textbf{in mkpar}\ (\textbf{fun}\ pid \rightarrow e)$$

The reduction will occur at the hole to first compute the value of x. The Γ_l^i context is used to define in which component i of a parallel vector the reduction is done, i.e., which processor i reduces its local expression. This context uses the Γ_l context which defines a local reduction on a processor i. Note that, in this way, the hole is always inside a parallel vector. For example, the following context: $\Gamma_l^i = \textbf{apply}\ v\ \langle v_0, e_1, \ldots, \Gamma_l \rangle$ and $\Gamma_l = \textbf{open}_l^{\mathbf{r}}\ []$ is used to define that the last processor first computes the argument of the $\textbf{open}_l^{\mathbf{r}}$ primitive.

Now we can reduce "in depth" in the sub-expressions. To define this deep reduction, we use the inference rules of the local context rule:

$$\frac{e\ /\ \{f_i\} \xrightarrow{i} e'\ /\ \{f_i'\}}{\{\mathcal{F}\}\ /\ \Gamma_l^i(e)\ /\ \{f\} \rightharpoonup \{\mathcal{F}\}\ /\ \Gamma_l^i(e')\ /\ \{f'\}} \quad \text{where} \quad \begin{cases} \{f\} = \{\{f_0\}, \ldots, \{f_i\}, \ldots, \{f_{p-1}\}\} \\ \{f'\} = \{\{f_0\}, \ldots, \{f_i'\}, \ldots, \{f_{p-1}\}\} \end{cases} \quad (7)$$

So, we can reduce the parallel vectors and the context gives the name of the processor where the expression is reduced. The global context rule is:

$$\frac{\{\mathcal{F}\} \,/\, e \,/\, \{f\} \overset{\bowtie}{\rightharpoonup} \{\mathcal{F}'\} \,/\, e' \,/\, \{f\}}{\{\mathcal{F}\} \,/\, \Gamma(e) \,/\, \{f\} \rightharpoonup \{\mathcal{F}'\} \,/\, \Gamma(e') \,/\, \{f\}} \quad (8)$$

We can remark that the context gives an order to evaluate an expression but not for the parallel vectors and this rule is not deterministic. It is not a problem because the BSλ-calculus is confluent [37]. We can also notice that our two kinds of contexts used in the rules exclude each other by construction because the hole in a Γ_l^i context is always in a component of a parallel vector and never for a Γ one. Thus, we have a rule to reduce global expressions and another one to reduce usual expressions within the parallel vectors and we have the following result of confluence:

THEOREM 5.1. *if* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^1\}/v_1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^2\}/v_2/\{f^2\}$ *then* $v_1 = v_2$, $\mathcal{F}^1 = \mathcal{F}^2$ *and* $f^1 = f^2$.

Proof. (Sketch of) The BSML language is known to be confluent [37]. With our two kinds of file systems, it is easy to see that a global rule never modifies a local file system and never a local rule modifies the global one. To be more formal, the global (resp. local) files are always the same before and after a local (resp. global) reduction. Thus, the global values are the same on all the processors as proof of confluent of the BSML language needed. All the δ-rules working on files are deterministic (local and global ones). So, the BSML language with parallel I/O features is confluent.□

We refer to appendix 9 for a full proof. Note that the semantics is not deterministic. Several rules can be applied at the same time, parallelism comes from context rules.

6. Formal Cost Model. A formal cost model can be associated to reductions in the BSML language. "cost terms" are defined and each rule of the semantics is associated to a cost rule on cost terms. Given the *weak call-by-value strategy*, i.e., arguments to functions and operators need to be values (see section 5), a program is always reduced in the "same way". As stated in [41], "Each evaluation order has its advantages and disanvatages, but strict evaluation is clearly superior in at least one area: ease of reasoning about asymptotic complexity". In this case, costs can be associated with terms rather than reductions. It is the way we choose to ease the discussion about the compositional nature of the cost model of our language and the cost of our I/O primitives.

6.1. Costs of the Parallel Operators. No order of reduction is given between the different components of a parallel vector and their evaluations are done in parallel. The cost in this case is independent from the order of reduction. We will not describe the costs of the evaluation of local terms, i. e., functional terms. They are the same as those of a strict functional language (OCaml for example) but we give the costs of the evaluation of global and I/O operations.

The cost model associated to our programs follows the EM^2-BSP cost model. We noted $\mathcal{C}(e)$ the cost term associated to an expression, $\mathcal{S}(v)$ the size in words of a serialized value v and $\oplus$ for the sum of cost with the following rules:

$$\begin{array}{rcl} c \oplus \phi c_0, \ldots, c_{p-1} \flat & = & \phi c + c_0, \ldots, c + c_{p-1} \flat \\ c^1 \oplus c^2 & = & c^2 \oplus c^1 \\ \phi c_0^1, \ldots, c_{p-1}^1 \flat \oplus \phi c_0^2, \ldots, c_{p-1}^2 \flat & = & \phi c_0^1 + c_0^2, \ldots, c_{p-1}^1 + c_{p-1}^2 \flat \end{array}$$

where c, c^1 c^2 are cost terms and $\phi c_0, \ldots, c_{p-1} \flat$ is the cost term associated to a parallel vector. Such rules say that the cost of replicate terms could be inside or outside a parallel vector cost term and when we have the cost term of a full-evaluated superstep, this cost could also be inside or outside a parallel vector cost term. This is not a problem because, using the BSP model of computation, at the end of a superstep, we take the maximal of the costs. $+$ and $\times$ are classical cost addition and multiplication using the EM^2-BSP parameters (g, l, r, G^l etc.). We also noted $\biguplus$ for the maximal cost of parallel vector cost terms with this rules: $\biguplus \phi c_0, \ldots, c_n, \ldots, c_{p-1} \flat = c_n$ if c_n is the maximal cost of the component of the parallel vector cost term. We also noted $\bigoplus_{i=0}^{p-1} h_i$ for the maximal of sent/received words. The EM^2-BSP costs of the parallel primitives are given in Figure 6.1. The cost of a program e is thus $\biguplus(\mathcal{C}(e))$ the maximal time for a processor to perform all the supersteps of the program. Let us explain such formal rules with more details and more "readable notations".

If the computational and I/O time for the evalution of the functional parameter e of **mkpar** is w_{all} (it is a replicate function and thus computed by all the processors) and if the sequential evaluation time of each

$$
\begin{array}{lcl}
\mathcal{C}(\textbf{mkpar } e) & \rightsquigarrow & \mathcal{C}(e) \oplus \phi\langle \mathcal{C}((f\ 0)), \ldots, \mathcal{C}((f\ (p-1)))\rangle \quad \text{if } e \stackrel{*}{\rightharpoonup} f \\
\mathcal{C}(\textbf{apply } e_1\ e_2) & \rightsquigarrow & \mathcal{C}(e_1) \oplus \mathcal{C}(e_2) \oplus \phi\langle \mathcal{C}((f_0\ v_0)), \ldots, \mathcal{C}((f_{p-1}\ v_{p-1}))\rangle \\
 & & \qquad \text{if } \left\{ \begin{array}{l} e_1 \stackrel{*}{\rightharpoonup} \langle f_0, \ldots, f_{p-1}\rangle \\ e_2 \stackrel{*}{\rightharpoonup} \langle v_0, \ldots, v_{p-1}\rangle \end{array} \right. \\
\mathcal{C}(\textbf{put } e) & \rightsquigarrow & \biguplus(\mathcal{C}(e) \oplus \phi\langle \sum_{j=0}^{p-1} \mathcal{C}((f_0\ j)), \ldots, \sum_{j=0}^{p-1} \mathcal{C}((f_{p-1}\ j))\rangle) \oplus (\bigoplus_{i=0}^{p-1} h_i) \times g \oplus l \\
 & & \qquad \text{where } \left\{ \begin{array}{l} \text{if } e \stackrel{*}{\rightharpoonup} \langle f_0, \ldots, f_{p-1}\rangle \\ \text{if } \forall i, j \in \{0, \ldots, p-1\}\ (f_i\ j) \stackrel{*}{\rightharpoonup} v_j^i \\ \text{and } h_i = \bigoplus(\sum_{j=0}^{p-1} \mathcal{S}(v_j^i), \sum_{j=0}^{p-1} \mathcal{S}(v_i^j)) \end{array} \right. \\
\mathcal{C}(\textbf{at } e_1\ e_2) & \rightsquigarrow & \biguplus(\mathcal{C}(e_1) \oplus \mathcal{C}(e_2)) \oplus (p-1) \times \mathcal{S}(v_n) \times g \oplus l \\
 & & \qquad \text{if } \left\{ \begin{array}{l} e_2 \stackrel{*}{\rightharpoonup} n \\ e_1 \stackrel{*}{\rightharpoonup} \langle v_0, \ldots, v_n, \ldots, v_{p-1}\rangle \end{array} \right.
\end{array}
$$

FIG. 6.1. *Costs of our parallel operators*

component of the parallel vector is $w_i + m_i$ (computational time and I/O time) then, the parallel evaluation time of the parallel vector is $\phi\langle w_{all} + w_0 + m_0, \ldots, w_{all} + w_{p-1} + m_{p-1}\rangle$, i.e, it is a local computation.

Provided the two arguments of the parallel application are parallel vectors of values, and if w_i (resp. m_i) is the computational time (resp. I/O time) of $(f_i\ v_i)$ at processor i, the parallel evaluation time of (**apply** $\langle f_0, \ldots, f_{p-1}\rangle\ \langle v_0, \ldots, v_{p-1}\rangle$) is $\phi\langle w_{all} + w_0 + m_0, \ldots, w_{all} + w_{p-1} + m_{p-1}\rangle$ where w_{all} is the computational and I/O time to create the two parallel vectors.

The evaluation of **put** $\langle f_0, \ldots, f_{p-1}\rangle$ requires a full superstep. Each processor evaluates the p local terms $(f_i\ j)$, $0 \leq j < p$ leading to p^2 sending values v_i^j (first phase of the superstep). If the value v_i^j of processor i is different from **None**, it is sent to processor j (communication phase of the superstep). Once all values have been exchanged, a synchronization barrier occurs. So, the parallel evaluation time is:

$$\max_{0 \leq i < p} (w_i + m_i + w_{all}) \oplus \max_{0 \leq i < p} (h_i \times g) \oplus l$$

where w_i (resp. m_i) is the computation time (resp. I/O time) of $(f_i\ j)$, h_i is the number of words transmitted (or received) by processor i and w_{all} is the computation time to create the parallel vector $\langle f_0, \ldots, f_{p-1}\rangle$.

The evaluation of a global projection (**at** $\langle v_0, \ldots, v_{p-1}\rangle\ n$) where n is an integer value also requires a full superstep. First the processor n sends the value v_n to all other processors and then a synchronization barrier occurs. The parallel evaluation time is thus the time to send this data, the time for compute n and the maximal local computation and I/O time to create the parallel vector $\langle v_0, \ldots, v_{p-1}\rangle$.

6.2. Cost of I/O operators. Our I/O operators have naturally some computational and I/O costs. We also made sure that arguments of the I/O operators be evaluated first (*weak call-by-value strategy*). As explained in the EM^2-BSP model, each transfer from (resp. to) the local external memory to (resp. from) the main memory has the cost $\lceil \frac{n}{D^l} \rceil \times G^l + \lceil \frac{n+1}{D^l B^l} \rceil \times O^l$ (resp. $\lceil \frac{n}{D^g} \rceil \times G^g + \lceil \frac{n+1}{D^g B^g} \rceil \times O^g$ for the global external memory) for n words. Note that, in the case of an empty file, the value to be read would be an empty value with an empty size. Thus the cost would just be the overhead. In this way, we have the cost of the "operating system I/O calls". Depending on whether the global file system is shared or not, the global I/O operators have different costs and some barrier synchronizations are needed (Figure 6.2).

Local operators are asynchronous operators. They belong to the first phase of a superstep. In the case of a distributed global file system, a global operator has the same cost as a local operator. But, in the case of global shared disks, global operators are synchronous operators because they modify the global behaviour of the EM^2-BSP computer. The two exceptions are **glo_output_value** and **glo_input_value** which are asynchronous global operators because only one process really has to write this replicate value (which is thus the same on each processor) or each processor read this value. The reading of this value could be done in any order. Different channels are positioned at different places in the file but read the same value for the same position. For example, opening a global file needs a synchronization because **glo_output_value** and **glo_input_value**

Operator	Cost
loc_open_in (resp. out)	constant time t^l_{or} (resp. t^l_{ow})
(**loc_output_value** v)	$\lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l$
loc_input_value	$\lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l$ where v is the readed value
loc_close_in (resp. out)	constant time t^l_{cr} (resp. t^l_{cw})
loc_delete	constant time t^l_d
glo_open_in	$\begin{cases} (p-1)\times g \ + \ t^g_{or} \ + \ l & \text{If global file system shared} \\ t^l_{or} & \text{Otherwise} \end{cases}$
glo_open_out	$\begin{cases} (p-1)\times g \ + \ t^g_{or} \ + \ l & \text{If global file system shared} \\ t^l_{ow} & \text{Otherwise} \end{cases}$
(**glo_output_value** v)	$\begin{cases} \lceil\frac{size(v)}{D^g}\rceil \times G^g + \lceil\frac{size(v)+1}{D^g B^g}\rceil \times O^g & \text{If shared} \\ \lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l & \text{Otherwise} \end{cases}$
glo_input_value	$\begin{cases} \lceil\frac{size(v)}{D^g}\rceil \times G^g + \lceil\frac{size(v)+1}{D^g B^g}\rceil \times O^g & \text{If shared} \\ \lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l & \text{Otherwise} \\ \text{and where } v \text{ is the readed value} \end{cases}$
glo_close_in	$\begin{cases} (p-1)\times g \ + \ t^g_{cr} \ + \ l & \text{If global file system shared} \\ t^l_{cr} & \text{Otherwise} \end{cases}$
glo_close_out	$\begin{cases} (p-1)\times g \ + \ t^g_{cw} \ + \ l & \text{If global file system shared} \\ t^l_{cw} & \text{Otherwise} \end{cases}$
glo_delete	$\begin{cases} (p-1)\times g \ + \ t^g_{d} \ + \ l & \text{If global file system shared} \\ t^l_{d} & \text{Otherwise} \end{cases}$
(**glo_copy** F f)	$\begin{cases} \lceil\frac{size(F)}{D^g}\rceil \times G^g + \lceil\frac{size(F)}{D^g B^g}\rceil \times O^g + \lceil\frac{size(F)}{D^l}\rceil \times G^l + \lceil\frac{size(F)}{D^l B^l}\rceil \times O^l + l & \text{If global file system shared} \\ (\lceil\frac{size(F)}{D^l}\rceil \times G^l + \lceil\frac{size(F)}{D^l B^l}\rceil \times O^l) \times 2 + size(F) \times g + l & \end{cases}$

FIG. 6.2. *Formal costs of our I/O operators*

are asynchronous operators and a processor could never write in a global file when another reads in this file or opens it in read mode. With this barrier of synchronization, all the processors open (resp. close) the file and they could communicate to each other whether they managed to open (resp. close) that file without errors or not. In this way, $p-1$ booleans are sent on the network and a global exception will be raised if there are any problems.

6.3. Formal Cost Composition. The costs (parallel evaluation time) above are context independents. This is why our cost model is compositional. The compositional nature of this cost model relies on the absence of nesting of parallel vectors (our static analysis enforces this condition [23]) and the fact of having two kinds of file systems. A global I/O operator which accesses a global file and which could make some communications and synchronizations never occurs locally. If the nesting was not forbidden, for a parallel vector v and a **scan** function, the following expression (**mkpar** (**fun** $i \rightarrow$ **if** $i=0$ **then** (**scan** e $(+)$ v) **else** v)) would be a correct one. The main problem is the meaning of this expression.

We said that (**mkpar** f) evaluates to a parallel vector such that processor i holds value (f i). In the case of our example, this means that processor 0 should hold the value of (**scan** e $(+)$ v). Since the semantics of the language is confluent, it is possible to evaluate (**scan** e $(+)$ v) locally. But in this case, processor 0 would not have all the needed values. We could choose that another processors broadcast there own values to processor 0 and then processor 0 evaluates (**scan** e $(+)$ v) locally. The execution time will not follow the formula

given by the above cost model because the broadcasting of these values need additional communications and a synchronization. Thus, we have additional costs which are context dependent. The cost of this expression will then depend on its context. The cost model will no be compositional. This preliminary broadcast is not needed if (**scan** e $(+)$ v) could be not under a **mkpar**. Furthermore, the above solution would imply the use of a scheduler for each processor to know, at every time, if the processor need the values of other processors or not. Such constraints make the cost formulas very difficult to write.

As explained above, if the global file system is shared, only one process has to actually write a value to a global file. In this way, if this value is different on each processor (case of a parallel vector of values) then processors would asynchronously write different values on a shared file and we will not be able to reconstruct this value. The confluence of the language would be lost. In the case of a distributed global file system, this problem does not occur because each processor writes the value on a different file system. Programs would not be portables because they would be architecture dependent. The compositional nature of the cost model is also lost because the final results would depend on the EM^2-BSP architecture and not on the program. This is why it is forbidden to write global values to keep safe the compositional nature of the cost model. Note that the semantics forbids a parallel operator or a parallel persistent operator to be used inside a parallel vector and also forbid a local persistent operator to be used outside a parallel vector.

7. Experiments.

7.1. Implementation. The glo_channel and loc_channel are abstract types and are implemented as arrays of channels, one channel per disk. The current implementation used the thread facilities of OCaml to write (or read) on the D-disks of the computers: we create D-threads which write (or read) on the D channels. Each thread has a part of the data represented as a sequence of bytes and write it in parallel with other threads. To do this, we need to *serialize* our values, i.e., transform our values into a sequence of bytes to be written on a file and decoded back into a data structure. The module Marshal of OCaml provides this feature.

In the case of global shared disks, one of the processors is selected to really write the value, in our first implementation, each of them in turn. To communicate booleans, we used the primitives of communication of BSML. A total exchange of the booleans indicates if the processors has well opened/closed the file or not. The global and the local file systems are in different directories that are parameters of the language. The global directory is supposed to be mounted to access to the shared disks or is in different directories in the case of a distributed global file system. Therefore, global operators accessed to the global directory and local operators accessed to the local directories. In the case of shared disks without local disks, for example, using the library in a sequential machine as a PC, local operators use the "pid" of the processor to distinguish the local files of the different processors.

7.2. Example of functions using our library. Our example is the classical computation of the *prefix* of a list. Here we make the hypothesis that the elements of the list are distributed on all the processes as files which contain sub-parts of the initial list. Each file is cut out on sub-lists with $\frac{D^l \times B^l}{s}$ elements where s is the size of an element. We now describe the algorithm. We first recal the sequential OCaml code part of our algorithm:

```
let isnc=function None→true | _→false

(* seq_scan_last:(α →α →α )→α →α list→α *α list*)
let seq_scan_last op e l =
 let rec seq_scan' last l accu = match l with
            []→(last,(List.rev accu))
     | hd::tl→(let new_last = (op last hd)
                  in seq_scan' new_last tl (new_last::accu))
  in seq_scan' e l []
```

where List.rev $[v_0; v_1; \ldots; v_n] = [v_n; \ldots; v_1; v_0]$. To compute the prefix of a list, we first locally compute the prefix of the local lists located on the local files. For this, we used the following code:

```
(* seq_scan_list_io:(α →α →α )→α →loc_name→loc_name→α  *)
let seq_scan_list_io op e name_in name_tmp=
 let cha_in =loc_open_in name_in in
```

```
 let cha_tmp=loc_open_out name_tmp in
 let rec seq_scan' last =
  let block=(loc_input_value cha_in) in
   if (isnc block) then last
    else let block2=(seq_scan_last op last (noSome block)) in
      loc_output_value cha_tmp (snd block2);
      seq_scan' (fst block2) in
 let res=seq_scan' e in
  loc_close_in cha_in;loc_close_out cha_tmp;res
```

The local file is opened as well as another temporary file. For all the sub-lists of the file, we compute the prefix and the last elements of these prefixes. Then, we write these prefixes to the temporary file and we close the two files. Second, we compute the parallel prefix of the last elements of each prefix of that file. Third, we add those values to the temporary prefixes.

```
(* add_last:(α →α →α )−α →loc_name→loc_name→unit *)
let add_last op e name_tmp name_out =
 let cha_tmp=loc_open_in name_tmp in
 let cha_out=loc_open_out name_out in
 let rec seq_add () =
  let block = (loc_input_value cha_tmp) in
   if (isnc block) then () else
    loc_output_value cha_out(List.map (op e)(noSome block));
    seq_add () in
 seq_add ();loc_close_in cha_tmp;
 loc_close_out cha_out; loc_delete name_tmp
```

The operating of this function is similar to seq_scan_list_io and the full function is thus the composition of the above functions.

```
(*scan:(α →α →α )→α →loc_name→loc_name→loc_name→unit par*)
let scan_list_direct_io op e name_in name_tmp name_out =
 let lasts=parfun (seq_scan_list_io op e name_in)
                                        (replicate name_tmp) in
 let tmp_values=scan_direct op lasts in
  parfun3 (add_last op) tmp_values
                  (replicate name_tmp) (replicate name_out)
```

For example of the use of global files, we give the code of the distribution of the sub-lists to the processors: for each block of the initial list, one processor writes it to its local file.

```
(* distribut:glo_name→loc_name→unit *)
let distribut name_in name_out =
 let cha_in=glo_open_in name_in in
 let cha_outs=parfun loc_open_in (replicate name_out) in
 let rec distri m =
  let block=glo_input_value cha_in in
   if (isnc block) then () else
    (apply2 (mkpar (fun pid→if pid=m then loc_output_value
                                  else (fun a b→()))))
          cha_outs (replicate (noSome block)));
          distri ((m+1) mod (bsp_p())) in
  distri 0;parfun loc_close_out cha_outs;glo_close_in cha_in
```

We have the following cost formula for the I/O scan-list version using a direct scan algorithm:

$$(p-1) \times s \times g + 4 \times N \times (B^l \times G^l + O^l) + 2 \times r \times N \times (D^l \times B^l) + T^1 + l$$

if we read sub-lists of the files by block of size $D^l B^l$ where s denotes the size in words of a element, N is the

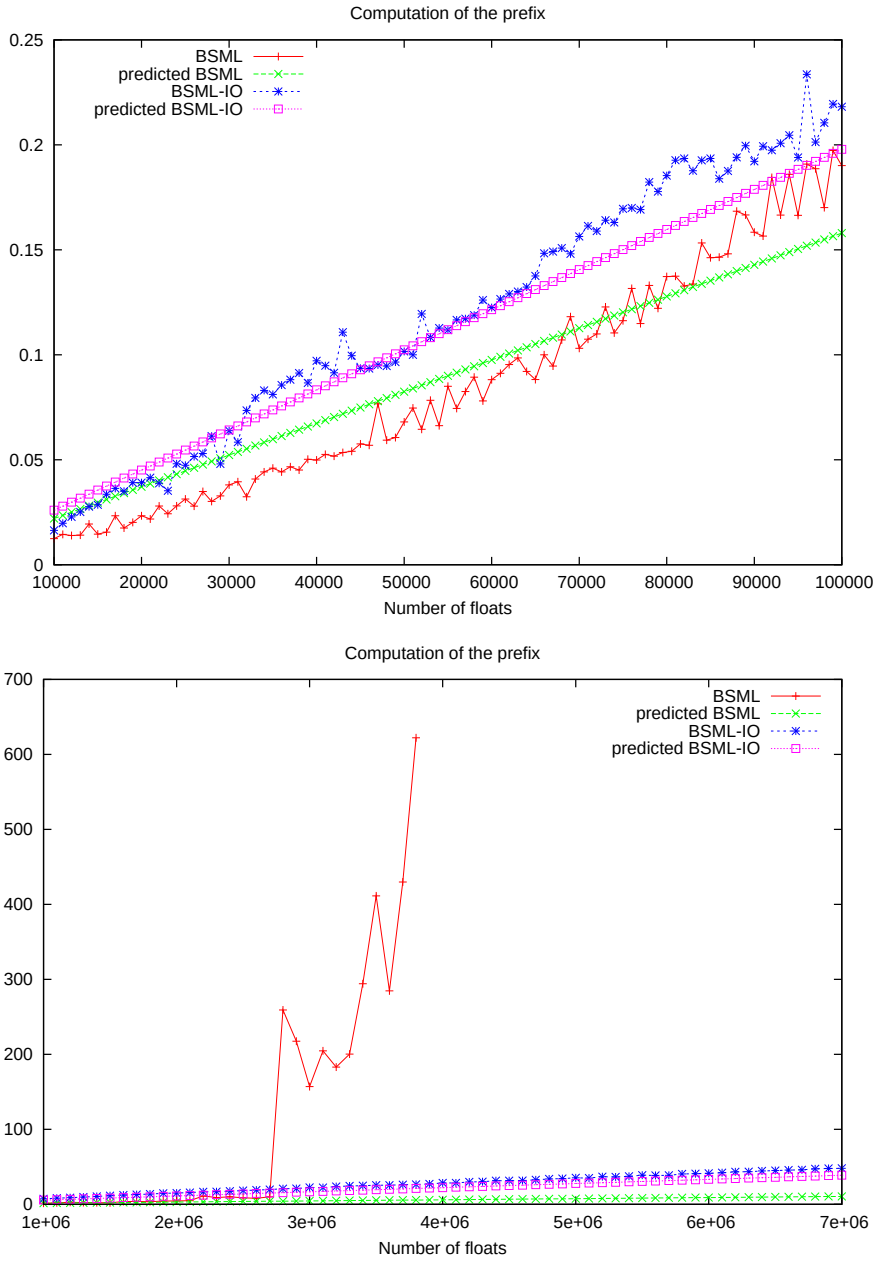


FIG. 7.1. *Benchmarks of prefix computations*

maximal length of a file on a process and where T^1 the time to open and close the files. We have the time to read the local files, write the temporary results of the temporary files, compute the local scan, read the local tempory files and write the final result in the final files. The cost formula of the distribution is:

$$\begin{cases} p \times N \times (\lceil \frac{D^l B^l}{D^g} \rceil \times G^g + \lceil \frac{D^l B^l}{D^g B^g} \rceil \times O^g) + N \times (B^l \times G^l + O^l) + 2 \times l + T^2 \\ \qquad\qquad \text{If shared global file system} \\ p \times N \times (B^l \times G^l + O^g) + N \times (B^l \times G^l + O^l) + T^2 \quad \text{Otherwise} \end{cases}$$

where T^2 the time to open and close the files. We have the time to read the data on the global file (read by block of size $D^l B^l$) and to write them on the local files. We also have two barriers of synchronization due to **glo_open_in** and **glo_close_in**. The cost formula for a global distributed file system is simpler than this

with shared disks but having a distributed file system makes the hypothesis that the global files are replicated on all the processors.

7.3. Benchmarks. Preliminary experiments have been done on a cluster with 6 Pentium IV nodes interconnected with a Gigabit Ethernet network to show a performance comparison between a BSP algorithm using only the BSMLlib and the corresponding EM^2-BSP algorithm using our library. The BSP algorithm reads the data from a global file and keeps them in the main memories. The EM^2-BSP algorithm distributed the data as in the above section. Figure 7.1 summarizes the timings. These programs were run 100 times and the average was taken. Only the local computation has been taken into account because the cluster do not have a true shared disk but a simulated shared disk using NFS. Therefore, the distribution of the data is very slow: G^g depends on g and the distribution of the two different algorithms takes approximately the same time.

The cluster has the following EM^2-BSP parameters:

p	=	6	nodes	D^l	=	1	bytes	D^g	=	1	bytes
r	=	469	Mflops/s	B^l	=	4096	bytes	B^g	=	4096	bytes
g	=	28	flops	G^l	=	1.2	flops	G^g	=	33.33	flops
l	=	22751	flops	O^l	=	100	flops	O^g	=	120	flops

using the MPI implementation of the BSMLlib and with 256 Mbytes of main memory per node. The BSP parameters have been obtained by using the bsmlprobe described in [5] and the I/O parameters have been obtained by using benchmarks as those of the Figure 4.2. The predicted performances using those parameters are also given. We have used floats as elements with $e = 0$, $\mathbf{op} = +$ and we have approximately 140 floats in one block and thus the lists are cut out on sub-lists with 140 elements.

For small lists and thus for a small number of data the overhead for the external memory mapping makes the BSML program outperform the EM^2-BSML one. However, once the main memory is all utilized, the performance of the BSML program degenerates (cost of the paging mechanism to have a virtual memory). The EM^2-BSML program continues "smoothly" and clearly outperforms the BSML code. Note that there is a step between the predictions of the performances and the true performances. This is due to the garbage collector of the OCaml language. In the ML family, the abstract machine manages the resources and the memory, unlike in C or C++ where the programmer has to allocate and de-allocate the data of the memory. Using I/O operators and thus a less naive algorithm achieved a scalability improvement for a big number of data.

8. Related Work. With few exceptions, previous authors focused on a uniprocessor EM model. The *Parallel Disk Model* (PDM) introduced by Vitter and Shriver [54] is used to model a two-level memory hierarchy consisting of D parallel disks connected to $v \geq 1$ processors via a shared memory or a network. The PDM cost measure is the number of I/O operations required by an algorithm where items can be transferred between internal memory and disks in a single I/O operation. While the PDM captures computation and I/O costs, it is designed for a specific type of communication network where a communication operation is expected to take a single unit of time, comparable to a single CPU instruction. BSP and similar parallel models capture communication and computational costs for a more general class of interconnection networks, but do not capture I/O costs. [8] presents an out-of-core parallel algorithm for inversions of big matrices. The algorithm only used broadcasts as primitive of communication with a cost as the BSP cost of a direct broadcast. The I/O costs are similar to ours: linear cost (and not constant cost) to read/write from/to the parallel disks.

Some other parallel functional languages like SAC [25], Eden [31] or GpH [49] offer some I/O features but without any cost model [30]. Parallel EM algorithms need to be carefully hand-crafted to work optimally and correctly in EM environments. I/O operators in SAC have been written for shared disks without formal semantics and the programmer is responsible for underterministic results of such operations. In parallel extensions of the Haskell language (web page `http://haskell.org`) like Eden and Gph, the safety and the confluence of I/O operators are ensured by the use of *monads* [56] and local external memories. Using shared disks is not specified in the semantics of these languages. These parallel languages also authorize processor to exchanged channels and give the possibility to read/write to/from them. It increases the expressiveness of the languages but decreases the cost prediction of the programs. Too many communications are hidden. It also makes the semantics difficult to write [3]. [24] presents a dynamic semantics of a mini functional language with a call-by-value strategy but I/O operators do not work on files. The semantics used a unique input entry (standard input) and a unique output. [18] develops a language for reasoning about concurrent pure functional I/O. They prove that under certain conditions the evaluation of this language is deterministic. But the files are only local files and no formal cost model is given.

In [12] the authors focused on optimization of some parallel EM sort algorithms using cache performances and the several layers of memories of the parallel machines. But they used low level languages and the large number of parameters in this model introduce a hardly tractable complexity. In [15] the authors have implemented some I/O operations to test their models but in a low level language and low level data. In the same manner, [26] describes an I/O library of an EM extension of its cost model which is a special case of the BSP model but also for a low level language. To our knowledge, our library is the first for an extension of the BSP model with I/O features called EM^2-BSP and for a parallel functional language with a formal semantics and a formal cost model. This cost model and our library could be used for large and parallel Data Base as in [2] where the authors used the BSP cost model to balance the communications and the local computations.

9. Conclusions and Future Works. The Bulk-Synchronous Parallel ML allows direct mode BSP programming and the current implementation of BSML is the BSMLlib library. But for some applications where the size of the problem is very significant, external memories are needed. In this paper we have presented an external memory extension of BSP model named EM^2-BSP and a way to extend the BSMLlib for I/O accesses in these external memories. The cost model of these new primitives and a formal semantics as persistent features have been investigated and some benchmarks have been done. This library is the follow-up to our work on imperative features of our functional data-parallel language [22].

There are several possible directions for future works. The first direction is the implementation of persistent primitives using special parallel I/O libraries as described in [29]. For example, low level libraries for shared RAID disks could be used for a fault tolerance implementation of the global I/O primitives.

A complementary direction is the implementation of BSP algorithms [13, 38, 45] and their transformations into EM^2-BSP algorithms as described in [16]. We will design a new library of classical programs as in the BSMLlib library to be used with large computational problems. We also have extended the model to include shared disks. To validate the cost model of these programs, we need a benchmark suite in order to automatically determine the EM parameters. This is ongoing work. We are also working on a result of simulation of a shared external memory as those of the main memory in the BS-PRAM of [48].

A semantic investigation of this framework is another direction of research. To ensure safety and a compositional cost model which allow cost analysis of the programs, two kinds of persistent primitives are needed, global and local ones. Such operators need occur in their context (local or global) and not in another one. We are currently working on a flow analysis [43] of BSML to avoid this problem statically and to forbid nesting of parallel vectors. Static cost analysis as in [51] is also another direction of research.

Acknowledgments The author wishes to thanks the anonymous referees of the Practical Aspects of High-Level Parallel Programming workshop (PAPP 2004), Frédéric Loulergue, Anne Benoï£¡ and Myztzu Modard for their comments.

REFERENCES

[1] *The Coq Proof Assistant (version 8.0)*. Web pages at coq.inria.fr, 2004.
[2] M. Bamha and M. Exbrayat, *Pipelining a Skew-Insensitive Parallel Join Algorithm*, Parallel Processing Letters, 13 (2003), pp. 317–328.
[3] J. Berthold and R. Loogen, *Analysing dynamic channels for topology skeletons in eden*, Tech. Rep. 0408, Institut für Informatik, Lübeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds.
[4] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*, Springer, 2004.
[5] R. Bisseling, *Parallel Scientific Computation. A structured approach using BSP and MPI*, Oxford University Press, 2004.
[6] G.-H. Botorog and H. Kuchen, *Efficient high-level parallel programming*, Theoretical Computer Science, 196 (1998), pp. 71–107.
[7] E. Caron, O. Cozette, D. Lazure, and G. Utard, *Virtual memory management in data parallel applications*, in HPCN Europe, 1999, pp. 1107–1116.
[8] E. Caron and G. Utard, *On the performance of parallel factorization of out-of-core matrices*, Parallel Computing, 30 (2004), pp. 357–375.
[9] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, D. E. Vengroff, and J. S. Vitter, *External-memory Graphs Algorithms*, in ACM-SIAM Symp on Discrete Algorithms, 1995, pp. 139–149.
[10] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner, *Performance issues of the parallel pam-crash code*, Supercomputer Applications and High Performance Computing, 11 (1997), pp. 3–11.
[11] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos, *Randomized External Memory Algorithms for Geometric Problems*, in ACM Annual Conf on Computational Geometry, 1998, pp. 259–268.
[12] C. Cérin and J. Hai, eds., *Parallel I/O for Cluster Computing (Hardback)*, Kojan Page Science, hermes spenton ed., 2002.
[13] F. Dehne, *Special issue on coarse-grained parallel algorithms*, Algorithmica, 14 (1999), pp. 173–421.

[14] F. Dehne, W. Dittrich, and D. Hutchinson, *Efficient external memory algorithms by simulating coarse-grained parallel algorithms*, Algorithmica, 36 (2003), pp. 97–122.
[15] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari, *Parallel virtual memory*, in 10th Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, 1999, pp. 889–890.
[16] ———, *Bulk synchronous parallel algorithms for the external memory model*, Theory of Computing Systems, 35 (2003), pp. 567–598.
[17] W. Dittrich and D. Hutchinson, *Blocking in Parallel Multisearch Problems*, Theory of Computing Systems, 34 (2001), pp. 145–189.
[18] M. Dowse and A. Butterfield, *A language for reasoning about concurrent functional I/O*, Tech. Rep. 0408, Institut für Informatik, Lübeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds.
[19] P. Ferragina and F. Luccio, *String search in coarse-grained parallel computers*, Algorithmica, 24 (1999), pp. 177–194.
[20] F. Gava, *Formal Proofs of Functional BSP Programs*, Parallel Processing Letters, 13 (2003), pp. 365–376.
[21] ———, *Parallel I/O in Bulk Synchronous Parallel ML*, in The International Conference on Computational Science (ICCS 2004), Part III, M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, eds., LNCS, Springer Verlag, 2004, pp. 339–346.
[22] F. Gava and F. Loulergue, *Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features*, in Parallel Computing: Software Technology, Algorithms, Architectures and Applications, Proceeding of the 10th ParCo Conference, G. Joubert, W. Nagel, F. Peters, and W. Walter, eds., Dresden, 2004, North Holland/Elsevier, pp. 95–102.
[23] ———, *A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting*, Future Generation Computer Systems, 21 (2005), pp. 665–671.
[24] A. Gordon and R. L. Crole, *A sound metalogical semantics for input/output effects*, Mathematical Structures in Computer Science, 9 (1999), pp. 125–188.
[25] C. Grelck and S.-B. Scholz, *Classes and objects as basis for I/O in SAC*, in Proceedings of IFL'95, Gothenburg, Sweden, 1995, pp. 30–44.
[26] J. Gustedt, *Towards realistic implementations of external memory algorithms using a coarse grained paradigm*, Tech. Rep. 4719, INRIA, 2003.
[27] G. Hains, *Parallel functional languages should be strict*, in Workshop on General Purpose Parallel Computing. World Computer Congress, B. Perhson and I. Simon, eds., vol. 1, IFIP, North-Holland, September 1994, pp. 527–532.
[28] J. Hill, W. McColl, and al., *BSPlib: The BSP Programming Library*, Parallel Computing, 24 (1998), pp. 1947–1980.
[29] H. Jin, T. Cortes, and R. Buyya, eds., *High Performance Mass Storage and Parallel I/O*, IEEE Press, wiley-interscience ed., 2002.
[30] P. T. K. Hammond and all, *Comparing parallel functional languages: Programming and performance*, Higher-order and Symbolic Computation, 15 (2003).
[31] U. Klusik, Y. Ortega, and R. Pena, *Implementing EDEN: Dreams becomes reality*, in Proceedings of IFL'98, K. Hammond, T. Davie, and C. Clack, eds., vol. 1595 of LNCS, Springer-Verlag, 1999, pp. 103–119.
[32] M. V. Kreveld, J. Nievergelt, T. Roos, and P. W. (editor), *Algorithms Foundations of Geographics Information Systems*, in International Symposium on High Performance Computing, no. 1340 in Lecture Notes in Computer Science, Springer, 1997.
[33] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon, *The Objective Caml System release 3.08*, 2004. web pages at www.ocaml.org.
[34] X. Leroy and M. Mauny, *Dynamics in ML*, Journal of Functional Programming, 3 (1993), pp. 431–463.
[35] Z. Li, P. H. Mills, and J. H. Reif, *Models and resource metrics for parallel and distributed computation*, Parallel Algorithms and Applications, 9 (1995), pp. 35–59.
[36] W. B. Ligon and R. B. Ross, *Beowulf Cluster Computing with Linux*, T. Sterling, mit press ed., November 2001, ch. PVFS: Parallel Virtual File System, pp. 391–430.
[37] F. Loulergue, G. Hains, and C. Foisy, *A Calculus of Functional BSP Programs*, Science of Computer Programming, 37 (2000), pp. 253–277.
[38] W. F. McColl, *Scalability, portability and predictability: The BSP approach to parallel programming*, Future Generation Computer Systems, 12 (1996), pp. 265–272.
[39] R. Milner, *A theory of type polymorphism in programming*, Journal of Computer and System Sciences, 17 (1978), pp. 348–375.
[40] K. Munagala and A. Ranade, *I/O Complexity of Graph Algorithms*, in ACM-SIAM Symposium on Discrete Algorithms, 1999, pp. 687–694.
[41] C. Okasaki, *Purely Functional Data-Structures*, Cambridge University Press, 1998.
[42] S. Pelagatti, *Structured Development of Parallel Programs*, Taylor & Francis, 1998.
[43] F. Pottier and V. Simonet, *Information Flow Inference for ML*, ACM Transactions on Programming Languages and Systems, 25 (2003), pp. 117–158.
[44] J. M. D. Rosario and A. Choudhary, *High performance I/O for massively parallel computers: Problems and prospects*, IEEE Computer, 27 (1994), pp. 59–68.
[45] J. F. Sibeyn and M. Kaufmann, *BSP-Like External-Memory Computation*, in Proc. 3rd Italian Conference on Algorithms and Complexity, vol. 1203 of LNCS, Springer-Verlag, 1997, pp. 229–240.
[46] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, *Questions and Answers about BSP*, Scientific Programming, 6 (1997), pp. 249–274.
[47] R. Thakur, E. Lusk, and W. Gropp, *I/O characterization of a portable astrophysics application on the ibm sp and intel paragon*, Tech. Rep. MCS-P534-0895, Argonne National Laboratory, October 1995.
[48] A. Tiskin, *The bulk-synchronous parallel random access machine*, Theoretical Computer Science, 196 (1998), pp. 109–130.
[49] P. Trinder and all., *GPH: An Architecture-independent Functional Language*, IEEE transactions on Software Engineerig, (1999).
[50] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), p. 103.

[51] P. B. Vasconcelos and K. Hammond, *Inferring cost equations for recursive, polymorphic and higher-order functional programs*, in IFL'02, LNCS, Springer Verlag, 2003, pp. 110–125.
[52] D. E. Vengroff and J. S. Vitter, *Supporting I/O-efficient Scientific Computation in TPIE*, in IEEE Symposium on Parallel and Distributed Computing, 1995.
[53] J. Vitter, *External memory algorithms*, in ACM Symp. Principles of Database Systems, 1998, pp. 119–128.
[54] J. Vitter and E. Shriver, *Algorithms for parallel memory, two -level memories*, Algorithmica, 12 (1994), pp. 110–147.
[55] J. S. Vitter, *External memory algorithms and data structures: Dealing with massive data*, ACM Computing Surveys, 33 (2001), pp. 209–271.
[56] P. Wadler, *Comprehending monads*, Mathematical Structures in Computer Science, 2 (1992), pp. 461–493.

Appendix A. Proof of the confluence.

LEMMA A.1. *If* $e/\{f_i\} \xrightarrow{i} e^1/\{f_i^1\}$ *and* $e/\{f_i\} \xrightarrow{i} e^2/\{f_i^2\}$ *then* $e^1 = e^2$ *and* $\{f_i^1\} = \{f_i^2\}$. *Proof.* By case of the rules of figures 5.1, 5.3 and by construction of rules (1), (3) and (5). □

LEMMA A.2. *If* $\{\mathcal{F}\}/e/\{f\} \overset{\bowtie}{\rightharpoonup} \{\mathcal{F}^1\}/e^1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \overset{\bowtie}{\rightharpoonup} \{\mathcal{F}^2\}/e^2/\{f^2\}$ *then* $e^1 = e^2$, $\{f\} = \{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By case of the rules of figures 5.2, 5.3 and by construction of rules (2), (4) and (6). □

LEMMA A.3. *If* $e = \Gamma_l^i(e_1)$ *then* $\nexists\ e_2$ *as* $e = \Gamma(e_2)$; *If* $e = \Gamma(e_1)$ *then* $\nexists\ e_2$ *as* $e = \Gamma_l^i(e_2)$. *Proof.* By definition, the hole [] is inside a parallel vector in the case of a Γ_l^i context and outside a parallel vector in the other case. By construction, contexts excluded each other. □

DEFINITION A.4. *We noted* $\underset{\bowtie}{\Rightarrow}$ *the reduction* $\rightharpoonup$ *only using the rule (8) and* $\underset{i}{\Rightarrow}$ *the reduction* $\rightharpoonup$ *only using the rule (7).*

LEMMA A.5. *If* $\{\mathcal{F}\}/\Gamma_l^i(e)/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}^1\}/\Gamma_l^i(e^1)/\{f^1\}$ *and* $\{\mathcal{F}\}/\Gamma_l^i(e)/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}^2\}\ /\Gamma_l^i(e^2)/\{f^2\}$ *then* $e^1 = e^2$, $\{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}\} = \{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By application of lemma A.1 and by definition of rule (7). □

LEMMA A.6. *If* $\{\mathcal{F}\}/\Gamma(e)/\{f\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}^1\}/\Gamma(e^1)/\{f^1\}$ *and* $\{\mathcal{F}\}/\Gamma(e)/\{f\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}^2\}/\ \Gamma(e^2)/\{f^2\}$ *then* $e^1 = e^2$, $\{f\} = \{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By application of lemma A.2 and by definition of rule (8). □

DEFINITION A.7. *We noted* $\{\mathcal{F}\}/e/\{f\} \underset{l}{\Rightarrow} \{\mathcal{F}\}/e_1/\{f'\}$ *the reduction* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\underset{i}{\Rightarrow}} \{\mathcal{F}\}/e_1/\{f'\}\ \forall\ i$ *and where* $\nexists\ e_2 \wedge \Gamma_l^i$ *as* $e_1 = \Gamma_l^i(e_2)$ *and where* e_2 *is not a value.*

LEMMA A.8. *If* $\Gamma_l^i(e_1) = \Gamma_l^j(e_2)$ *and* $\{\mathcal{F}\}/\Gamma_l^i(e_1)/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}\}/\Gamma_l^i(e_3)/\{f^3\}$ *and* $\{\mathcal{F}\}/\Gamma_l^j(e_2)/\{f\} \underset{j}{\Rightarrow} \{\mathcal{F}\}/\Gamma_l^j(e_4)/\{f^4\}$ *then* $\exists\ \Gamma'^j_l \wedge \Gamma'^i_l$ *as* $\Gamma_l^i(e_3) = \Gamma'^j_l(e_2)$ *and* $\Gamma_l^j(e_4) = \Gamma'^i_l(e_1)$ *where* $\{\mathcal{F}\}/\Gamma'^j_l(e^2)/\{f^3\} \underset{j}{\Rightarrow} \{\mathcal{F}\}/\Gamma'^j_l(e_5)/\{f^5\}$ *and* $\{\mathcal{F}\}/\Gamma'^i_l(e_1)/\{f^4\} \underset{i}{\Rightarrow} \{\mathcal{F}\}/\ \Gamma'^i_l(e_6)/\{f^6\}$ *and where* $\Gamma'^j_l(e_5) = \Gamma'^i_l(e_6)$ *and* $\{f^5\} = \{f^6\}$.
Proof. It is easy to see that a $\underset{i}{\Rightarrow}$ reduction only modify an expression of the i^{th} component of a parallel vector and the i^{th} file system. Such reduction is determinist by lemma A.5 and thus we have that if two reductions appear in two different components of a parallel vector then such reductions could be done in any order and give the same final result. □

LEMMA A.9. *If* $\{\mathcal{F}\}/e/\{f\} \underset{l}{\Rightarrow} \{\mathcal{F}\}/e_1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \underset{l}{\Rightarrow} \{\mathcal{F}\}/e_2/\{f^2\}$ *then* $e_1 = e_2$ *and* $\{f^1\} = \{f^2\}$.
Proof. By induction on the two reduction $\underset{l}{\Rightarrow}$ and using lemma A.8 to "re-stick" together different paths of the derivations: parallel reductions could be done in any order. □

DEFINITION A.10. $\Rightarrow = \underset{\bowtie}{\Rightarrow} \cup \underset{l}{\Rightarrow}$

LEMMA A.11. *If* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^1\}/v_1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^2\}/v_2/\{f^2\}$ *then* $v_1 = v_2$, $\{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By induction of the derivation and by using lemma A.3: for the two inductive cases, we have the two following cases: if $\{\mathcal{F}'\}/e'/\{f'\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}''\}/e''/\{f'\}$ then the reduction is deterministic by lemma A.6 else $\{\mathcal{F}'\}/e'/\{f'\} \underset{l}{\Rightarrow} \{\mathcal{F}'\}/e''/\{f''\}$ and then the reduction is also deterministic by lemma A.9. □

LEMMA A.12. *If* $\{\mathcal{F}\}/e/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}\}/e_1/\{f^1\}$ *then* $\{\mathcal{F}\}/e_1/\{f^1\} \overset{*}{\underset{i}{\Rightarrow}} \{\mathcal{F}\}/e_2/\{f^2\}$ *and where* e_2 *is a value*

or $\{\mathcal{F}\}/e_2/\{f^2\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}'\}/e_3/\{f^2\}$.

Proof. By induction and using lemma A.3 for each steps of the derivation. □

LEMMA A.13. *if* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}'\}/v/\{f'\}$ *then* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}'\}/v/\{f'\}$. *Proof.* By induction of the derivation. If the rule (8) is used, we are in the case of a global reduction and then we have a $\underset{\bowtie}{\Rightarrow}$ reduction. Else if the rule (7) is used, we are in the case of a local reduction and we have by lemma A.12 that we have a $\underset{l}{\Rightarrow}$ reduction. □

THEOREM A.14. ***confluence of the semantics***

Proof. if $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^1\}/v_1/\{f^1\}$ and $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^2\}/v_2/\{f^2\}$ then $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^1\}/v_1/\{f^1\}$ and $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^2\}/v_2/\{f^2\}$ by lemma A.13 and then $v_1 = v_2$, $\{f^1\} = \{f^2\}$ and $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$ by lemma A.11. □

Edited by: Frédéric Loulergue
Received: June 3, 2004
Accepted: June 5, 2005