Parallel I/O in Bulk-Synchronous Parallel ML

Frédéric Gava

LACL, University Paris XII, Créteil, France
gava@univ-paris12.fr

Abstract. Bulk Synchronous Parallel ML or BSML is a functional data-parallel language for programming bulk synchronous parallel (BSP) algorithms. The execution time can be estimated and 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. We present here a library of I/O features for BSML and its cost model.

1 Introduction

External memory (EM) algorithms 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. Important applications fall into this category ([12] 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. The combination of I/O computing, on multiple disks, with multiprocessor parallelism is a challenge for the "Large-Scale" computing community. Bulk-Synchronous Parallel ML or *BSML* is an extension of ML for programming *Bulk-Synchronous Parallel* (BSP) algorithms as functional programs associated with a compositional cost model. BSP computing is a parallel programming model introduced by Valiant [11] to offer a high degree of abstraction like PRAM models. Such algorithms offer portable, predictable and scalable performances on a wide variety of architectures ([8] for a survey). BSML expresses them with a small set of primitives taken from the *confluent* BSλ-calculus. Those operations are implemented as a parallel library (`http://bsmllib.free.fr`) for the functional programming language Objective Caml (`http://www.ocaml.org`).

Parallel disk I/O has been identified as a *critical component* of a suitable *high performance* computer. [2] 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. To take advantage of these new results, we have to extend the BSML language with parallel I/O features for programming this new kind of algorithms. This paper describes our first work in this direction. The remainder of this paper is organized as follows. In Section 2 we briefly present the BSML language. In Section 3 we introduce the EM-BSP model and the problems that appear in BSML. We then give in Section 4 the new primitives for BSML, the associated cost model and an example. We discuss related work and conclude (section 5). This paper is an extended abstract of a technical report which can be found at `http://www.univ-paris12.fr/lacl` where more details are given.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 331–338, 2004.

2 Functional Bulk-Synchronous Parallel ML

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 for a *synchronization barrier*. In this model, a parallel computation is subdivided in *super-steps*, at the end of which a barrier synchronization and a routing is performed. Hereafter all requests for data were posted during a preceding super-step are fullfilled.

There is currently no implementation of a full BSML language but rather a partial implementation as a library (the `BSMLlib` library) for Objective Caml. In particular, it offers the function `bsp_p:unit->int` such as the value of `bsp_p()` is p, the *static number* of processes of the parallel machine. There is also an *abstract polymorphic* type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. Our static analysis enforces this restriction [3]. The BSML parallel constructs operate on parallel vectors. Parallel vectors are created by:

```
mkpar: (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$ (`(mkpar f)`$=\langle(f\ 0),\ldots,(f\ (p-1))\rangle$. These values are said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global* (which is similar on 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 with:

```
apply: ('a -> 'b) par -> 'a par -> 'b par
```

so that `apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. The communication and synchronization phases are expressed by:

```
put:(int->'a option) par -> (int->'a option) par
```

where `'a option` is defined by: `type 'a option = None | Some of 'a`. 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`. Expression $(*)$ evaluates to a parallel vector containing a function `fd`$_i$ of delivered messages on every process. 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 global conditional, omitted here for the sake of conciseness.

3 External Memories in BSML

3.1 The EM-BSP Model

In the BSP model, the performance of a parallel computer is characterized by only three parameters (expressed as multiples of the local processing speed): p the number of processors, l the time required for a global synchronization and g the time for collectively delivering a 1-relation (communication phase where

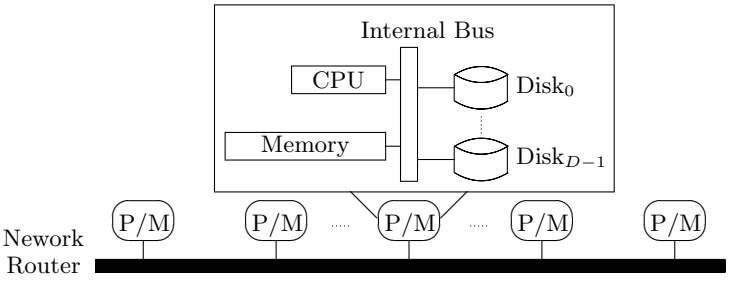

Fig. 1. A BSP Computer with External Memories

every processor receives/sends at most one word). The network can deliver an h-relation in time gh for any arity h.

[2] extended the BSP model to include secondary local memories. The basic idea is very simple and it is illustrated in Figure 1. Each processor has, in addition to its local memory, an EM in the form of a set of *disks*. Modern computers typically have several layers of memory which include main memory and caches as well as disks. We restrict ourselves to the two-level model because the speed difference between disk and main memory is much more signifiant than between the other layers of memory. This idea is applied to extend the BSP model to its EM version **EM-BSP** by adding the following parameters to the standard BSP parameters: M is the local memory *size* of each processor, D the number of *disk drives* of each processor, B the *transfer block size* of a disk drive and 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 numbers of drives and memory sizes for each processor). The disk drives of each processor are denoted by $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. Each processor can use all its D disk drives concurrently, and transfer $D \times B$ items from the local disks to its local memory in a single I/O operation being at cost G (thus, we do not deal with the intricacies of blocked secondary memory accesses). Each processor is assumed to be able to store in its local memory at least one block from each disk at the same time, i.e., $DB << M$. Like computation on the BSP model (for the sake of conciseness, we refer to [8] for more details), the computation on the EM-BSP model proceeds in a succession of super-steps and it allows multiple I/O operations during the computation phase of the super-step.

3.2 Problems by Adding I/O in BSML

The main problem in adding external memory (and so I/O operators) to BSML is to keep safe the fact that in the global context, the values on each processor are the same. For example, take the following expression:

```
let our_channel= open_in "file.txt" in
 let our_value=(input_value our_channel)) in ...
```

It is not true that the files (or the associate *channel*) on each processor contain the same value and in this case, each processor reads on its secondary memory a different value. If this expression had been evaluated with the `BSMLlib` library, and we would have obtained an incoherent result (and a crash of the BSP machine). Another problem come from *side-effects* that can occur on each processor:

```
let a=mkpar(fun i->if(i=0)then(open_in "file.txt");skip else skip)
  in (open_out "file.txt")
```

where only the first processor has opened a file in read mode and after, each processor opened the same file in write mode except the first processor: the file has already been open and we have an incoherent result. Our solution is to add two kinds of files: global and local ones. In this way, we also add two kinds of I/O operators. Local I/O operators do not have to occur in the global context (also global I/O operators do not have to occur locally) and the global files need to be the same on each node (in a *shared disks* or as a copy of the files on each process). An advantage having shared disks is in the case of some algorithms (as those which sort) when we have only one file (the list of data to sort) at the beginning of the program and one file (the sorted data) 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 new parameters of the EM-BSP machine: we have 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. For example, when the program either created or deleted a file because it seems to be impossible (un-deterministic) for a process to create a file in the global file system if at the same time another process deleted it. On the other hand, reading (resp. writing) values from files do not need any synchronization (only one of the processors need to really write the value on the shared disks). In the second case, all the files are distributed and no synchronization is needed (each processor read/write/delete etc. in its own file system) but at the beginning, the global files systems need to be empty. By this way, a new *ratio* (G^g) is needed which could be G is their is no shared disks. We supposed to have the same D and B for the shared disks (if not we can simulating these by cut the shared disks and change the G^g constant).

4 New Primitives and Their Costs

4.1 New Primitives

In this section we describe the core of our library, i.e, the minimal set of functions for programming EM-BSP algorithms. This library will be incorporated in the next release of the `BSMLlib`. As in the `BSMLlib` library, we used MPI and we have

functions to access to the EM-BSP parameters of the underlining architecture. In particular, if offers the functions `embsp_D:unit->int` which gives the number of disks and `is_global_shared: unit->bool` which gives if the global file system is shared or not. Since having two file systems, we need two abstract types of input channels and output channels: `glo_in_channel` (resp. `loc_in_channel`) and `glo_out_channel` (resp. `loc_out_channel`) to read or write in a global (resp. local) file. Therefore, we can open the named files for writing, and return a new output channel on that file, positioned at the beginning of the file. For this, we have two kinds of functions for global and local files:

```
glo_open_out : string -> glo_out_channel
loc_open_out : string -> loc_out_channel
```

The file is truncated to zero length if it already exists. It is created if it does not already exist. Raise `Sys_error` if the file could not be opened. In the same manner, we have two functions for opening a named file in read mode which returns a new input channel positioned at the beginning of the file. Now, with our channel, we can read and write values to the files. To do this, we need to "*serialize*" our values, i.e. transform our values to be written on a file: the module Marshal of the Objective Caml language provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file and can then be read back, possibly by another process. 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 : glo_out_channel -> 'a -> int
loc_output_value : loc_out_channel -> 'a -> int
```

which return the number of I/O operations used to write the values. The object can be read back, by the read functions:

```
glo_input_value : glo_in_channel -> int * 'a
loc_input_value : loc_in_channel -> int * 'a
```

which read the representation of a structured value and return the corresponding value with the number of I/O operations that have been done (we refer to [7] in order to have type safe values in channel and read it). To write (or read) on the D-disks of the machines, we used the thread facilities of Objective Caml: we create D-threads which write (or read) on the D-disks. The last primitive copies a local file from a processor to the global files system:

```
glo_copy : int -> string -> string -> unit par
```

and could be used at the end of the BSML program to copy the result to the global file system. It is not a communication primitive because this method has a more expensive cost than any communication primitive. As in any programming language, we also have some functions close channels, to set the current writing/reading position for channel, to return the total length (number of characters) of the given channel, or to return the total size, in characters of a value if it would be serialized to be written on a disk (and, thus, have the number of I/O operations needed). But for the sake of conciseness, we did not present them.

operator	cost
`loc_open_in`	constant time t_{or}^{l}
(`loc_output_value` v)	$G \times \lceil \frac{size(v)}{DB} \rceil$
`glo_open_in`	$\begin{cases} p \times t_{or}^{g} + l & \text{If shared global file system} \\ t_{or}^{l} & \text{Else} \end{cases}$
(`glo_output_value` v)	$\begin{cases} G^{g} \times \lceil \frac{size(v)}{DB} \rceil & \text{If shared global file system} \\ p \times G \times \lceil \frac{size(v)}{DB} \rceil & \text{Else} \end{cases}$
(`glo_copy` $file$)	$\begin{cases} \lceil \frac{size(file)}{DB} \rceil \times (G + G^{g}) + l & \text{If shared global file system} \\ \lceil \frac{size(file)}{DB} \rceil \times 2 \times G + size(file) \times g + l & \text{Else} \end{cases}$

Fig. 2. Cost of some operators

4.2 Formal Cost Model

Given the *weak call-by-value strategy*, a program is always reduced in the same way. In this case, costs can be associated to the parallel and I/O operators. The cost model associated to our programs follows our extention of the EM-BSP cost model. If the sequential evaluation time of each component of the parallel vector is $w_i + m_i$ (computional time and local I/O time), the parallel evaluation time of the parallel vector is $\max_{0 \leq i < p} w_i + \max_{0 \leq i < p} m_i$. Provided the two arguments of the parallel application are vectors of values, the parallel evaluation time of (**apply** $\langle f_0, \ldots, f_{p-1} \rangle$ $\langle v_0, \ldots, v_{p-1} \rangle$) is $\max_{0 \leq i < p} w_i + \max_{0 \leq i < p} m_i$ where w_i (resp. m_i) is the computational time (resp. I/O time) of $(f_i\ v_i)$ at processor i. To evaluate **put** $\langle f_0, \ldots, f_{p-1} \rangle$, first each processor evaluates the p local terms $(f_i\ j)$, $0 \leq j < p$ leading to p^2 sending values v_i^j. Once all values have been exchanged, a synchronization barrier occurs. At the beginning of this second super-step, each processor i constructs the function from the v_j^i received values. So, the parallel evaluation time of **put** $\langle f_0, \ldots, f_{p-1} \rangle$ is:

$$\underset{0 \leq i < p}{max}\ w_i^1 + \underset{0 \leq i < p}{max}\ h_i \times g + \underset{0 \leq i < p}{max}\ m_i + \underset{0 \leq i < p}{max}\ w_i^2 + l$$

where w_i^1 (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_i^2 is the computation time at processor i to constructing the result function from the v_j^i values. Our I/O operators have naturally some I/O costs and some computational time. We also provided that the arguments of the I/O operators have been evaluated first (italweak call-by-value strategy). As explained in the EM-BSP model, each transfer from (resp. to) the local files to (resp. from) its local memory has the cost G for DB items and depending if the global files system is shared or not, the global I/O operators have different costs and some barrier synchronisations is needed. The Figure 2 gives the costs of some selected operators (for more details, see the technical report). The cost (parallel evaluation time) above are context independent. 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 [3]) and the fact of having two kinds of file systems: a global I/O operator which access to a global file (and could make some communications or synchronization) never occurs locally.

4.3 Example

Our example is the classical *reduction* of lists. Each processor performs a local reduction, then sends its partial results to the following processors and finally localy reduces the partial results with the sended values:

```
em_scan_list (+) <[1;2], [3;4]> = <[1;1+2], [1+2+3, 1+2+3+4]>
```

for a reduction of two processors. But to take advantage of the disks and I/O operators, we suppose having large lists on each processor (6 billions of elements). These lists are supposed to be in a file on each processor and they are cutted out on sub-lists of sizes DB. The final result would be a file on each processor which contain sub-lists of size DB. Preliminary experiments on a cluster of PCs (7 bi-Pentium III with 512MB of RAM and ethernet network, see technical report) has been done to show a performance comparison between a BSP algorithms using only the `BSMLlib` and the corresponding EM-BSP code using our library to have some parallel virtual memories. For small lists, the overhead for the external memory mapping makes the BSML program outperform the EM-BSML one. However, once the main memory is all utilized, the performance of the BSML program degenerates (cost of the paging mechanism). The EM-BSML program continues "smoothly" and clearly outperforms the BSML code.

5 Conclusions and Future Works

With few exceptions (e.g. [12]), previous authors focused on a uniprocessor EM model. The *Parallel Disk Model* (PDM) introduced by Vitter and Shriver [12] 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 with 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. Some other parallel functional languages like SAC [4], Eden [6] or GpH [5] offer some I/O features but without any cost model, and parallel EM algorithms need to be carefully hand-crafted to work optimally and correctly in EM environments. In [1], the authors have implemented some I/O operations to test their models but in a low level language and low level data. To our knowledge, our library is the first for an extension of the BSP model with I/O features (called EM-BSP) and for a parallel functional language with a formal cost model.

The Bulk Synchronous Parallel ML allows direct mode Bulk Synchronous Parallel 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 memory is needed. We have presented in this paper an extention of BSP model named the EM-BSP for external memory and how to extend the `BSMLlib` for I/O access in this external memory. The cost model of these

new operators and a formal semantics (see technical report) have been investigated. This library is the continuity of our work about imperative and persistant features on our functional data-parallel language. To ensure safety and the compositional cost model, two kinds of I/O operators are needed (global and local ones) and those operators need not occur in another context (local or global). We are currently working on a flow analysis [9] for BSML to avoid this problem statically and to forbid nesting of parallel vectors. We are also working on the implementation of BSP algorithms [8] [10] and their transformations into EM-BSP algorithms as described in [2] to have a new library of classical programs as in the `BSMLlib` library to be used with large computational problems.

Acknowledgments. The authors wish to thank the anonymous referees for their comments. This work is supported by a grant from the French Ministry of Research and the ACI Grid program, under the project CARAML (www.caraml.org).

References

1. F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Parallel virtual memory. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 889–890, Baltimore, MD, 1999.
2. F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, 35:567–598, 2003.
3. F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In *PaCT 2003*, LNCS, pages 215–229. Springer Verlag, 2003.
4. C. Grelck and Sven-Bodo Scholz. Classes and objects as basis for I/O in SAC. In *Proceedings of IFL'95*, pages 30–44, Gothenburg, Sweden, 1995.
5. P.W. Trinder K. Hammond and all. Comparing parallel functional languages: Programming and performance. *Higher-order and Symbolic Computation*, 15(3), 2003.
6. U. Klusik, Y. Ortega, and R. Pena. Implementing EDEN: Dreams becomes reality. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of IFL'98*, volume 1595 of *LNCS*, pages 103–119. Springer-Verlag, 1999.
7. X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1994.
8. W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.
9. F. Pottier and V. Simonet. Information flow inference of ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
10. J. F. Sibeyn and M. Kaufmann. BSP-Like External-Memory Computation. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *LNCS*, pages 229–240. Springer-Verlag, 1997.
11. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.
12. J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory, two -level memories. *Algorithmica*, 12(2):110–147, 1994.