Implementation of data-parallel skeletons: a case study using a coarse-grained hierarchical model

Chong Li[*]
LACL, Université Paris-Est
and EXQIM S.A.S.

Frédéric Gava[†]
LACL, Université Paris-Est

Gaétan Hains[‡]
LACL, Université Paris-Est
and EXQIM S.A.S.

ABSTRACT

Writing parallel programs is known to be notoriously difficult. Often programmers do not want to reason about message-passing algorithms and only want to combine existing high-level patterns to produce their parallel program. This is the algorithmic skeletons approach to parallel programming. It improves reliability and clarity of source code. But skeletons can be insufficient when complicated communication schemes are needed. Expressing skeletons in a more general and low level language in the form of a library seems to be a good compromise between simplicity and expressive power. In this article, we present a coarsed-grained implementation using a hierarchical model of a set of data-parallel skeletons. Programming experiments and benchmarks complete the article.

KEYWORDS: Skeletons, hierarchical BSP (Bulk-Synchronous Parallelism).

1. INTRODUCTION

In the context of "Think Parallel or Perish", parallel code should be the norm. But programmers are not always able to manipulate low-level routines [8] or threads [13] without introducing many bugs such as deadlocks, livelocks, non-determinism *etc.* In fact, many parallel programs are not as unstructured as they appear and could be considered as compositions of parallel patterns/structures. This observation is behind the BSP [2] (Bulk-Synchronous Parallelism) and algorithmic skeletons [4] paradigms. Skeletal (resp. BSP) programming proposes that such patterns be abstracted and provided as a programmer's toolkit with specifications that transcend architectural variations but with implementations that recognize them to enhance performance. But sadly these structures are rarely used and one reason for this situation is their lack of expressive power. They often lack a sufficiently wide set of patterns for practical and efficient programming.

That makes the design of new and robust parallel programming languages an important area of research. Creating such a language involves a tradeoff between predictability and efficiency and the abstraction of parallel features to make programming safer and easier. SGL is the Scatter/Gather (data-parallel) Programming Language that we introduce as basis for programming skeletons and executing them on hierarchical or heterogeneous architectures with deterministic, non-blocking semantics and predictable performance.

Skeleton languages are generally defined by introducing a limited set of parallel patterns to be composed in order to build easily a full parallel application [4]. Even if the implementation is less efficient compared to a dedicated skeleton language (or a MPI send/receive implementation [6]), an implementation using SGL has two advantages. First, it allows skeleton programming when it is possible and normal SGL programming when no pattern is possible for the distribution of the data and of the computations. Second, some typical data-parallel computations can be easilly written using skeletons; it is thus a good example for benchmarking our methodology and compare the performances of SGL to other languages.

The next section formally define SGL's core features in their syntax, semantics, abstract parallel machine and performance prediction model with BSP-like parameters. We then propose a small library of standard algorithmic skeletons that are both easily and efficiently programmed in SGL, and directly applicable to such higher-level algorithms as FFT and triangular

[*] *chong.li@exqim.com*
[†] *frederic.gava@univ-paris-est.fr*
[‡] *gaetan.hains@univ-paris-est.fr*

system solving. The next sections detail an experimental implementation of those SGL programs, performance tests, scalability and their comparison with the performance prediction model. The final section analyzes our results and compares them with existing related works. It also summarizes ongoing and future work in the direction of a complete SGL library based on the LLVM programming system [12].

2. A HIERARCHICAL LANGUAGE

2.1. Abstract machine

The flat view of a parallel machine as a set of communicating sequential machines remains useful but is more and more incomplete. For example, GP-GPU processors have a master-worker architecture. With these issues in mind, Valiant introduced Multi- BSP [18] a multi-level variant of the BSP model[1] and showed how to design scalable and predictable algorithms for it. The main new feature of Multi-BSP is its hierarchical nature with nested levels that correspond to physical architectures' natural layers.
We now summarize our SGL, a programming model for Multi-BSP adaptable to heterogeneous systems. It assumes a tree-structured machine and uses only the following parallel primitives: scatter (to send data from master to workers), pardo (to request asynchronous computations from the workers) and gather (to collect data back to the master).
The authors have previously studied SGL only from the point of view of a language for implementing BSP primitives, with predictable performance and as an imperative language. The present paper is the first time SGL is applied to realistic algorithmic skeletons so the only content that has been published before is the language description *without* the discussion and formalization of horizontal communication. A detailed description of SGL can be found in [14] with examples that are nearly trivial compared to the skeletons analyzed here: reduction and parallel scan.

The SGL model assumed a set of sequential processors composed of a computation element ("core") and memory unit. The processors are arranged in a tree structure with the root being called a "master" and its children that are either masters themselves or leaf-"workers". The number of worker-children is not limited so that the BSP/PRAM concept of a flat p-vector of processors is easily recovered in SGL. Different forms are possible: one worker without master (a sequential machine), one master with p workers as children (e.g. a BSP computer where the non-localized parts of the SPMD code realize the master) or a general hierarchical machine. An SGL system has one and only one root-master. A master coordinates its children. A worker is controlled by one and only one master and all communication is between a master and its children.

[1]Valiant's BSP — Bulk-Synchronous Parallel programming — model is a bridging model between abstract execution and concrete parallel systems; It allows portable and scalable performance prediction for parallel programs. Its initial assumtions on the architecture were a homogeneous set of asynchronous processors with local memory, a completely connected communication network and a device for global synchronization.

An SGL program is composed of a sequence of super-steps, each one having 4 phases: (1) a scatter (comm.) phase initiated by the master; (2) an asynchronous computation phase performed by the children (this can also be a super-step, recursively); (3) a gather (comm.) phase centered on the master; (4) a local computation phase on the master. An SGL program is usually recursive: if node is a master and parallelism is needed then split input data into blocks; scatter them to children; process data blocks in children (recursively in parallel); gather children's results; else compute on local data directly. The *cost* of a program execution estimates its computational resources (usually time). The cost of a sequence of super-steps is a sum and so is the sum of each one's 4 phases. An SGL cost is almost exact a BSP cost with the following generalizations: (a) phases (1) and (3) have different *gap* (communication time per word) factors, one downward $g_\downarrow$ for scattering and one upward $g_\uparrow$ for gathering with proportionality factors $k_\downarrow, k_\uparrow$ that are machine dependent; (b) both communication phases also incur a fixed cost l called the latency parameter; (c) the system is in general heterogeneous so each computing element has its own computation speed parameter, but for simplicity we avoid a detailed description of this factor here; (d) the cost of a recursive super-step follows recursively the same formula with a worker being the master of its sub-system.

In general the cost formulae are $Cost_{Master} = \max_{i=1}^{p}(Cost_{child_i}) + w_0 \times c_0 + k_\downarrow \times g_\downarrow + k_\uparrow \times g_\uparrow + 2l$ and $Cost_{Worker} = w_i \times c_i$, which clearly covers the possibility of a heterogeneous architecture. The cost formula for $Cost_{Master}$ uses $Cost_{child_i}$ which refers either to the i-th sub-system with the same equation, or to $Cost_{Worker}$ if the i-th subsystem is a worker (leaf). More often, but not necessarily, we have symmetric communication: $Cost_{supstep} = w \times c + [\max_{i=1..p}(Cost_{chd_i}) + (k_\downarrow + k_\uparrow) \times g + 2l]$.

In summary we have a BSP-like cost model which treats additively the three components of execution time namely local computation, communication delay and synchronization. The first one is a maximum over sibblings executing concurrently. Communication is linearly proportionnal to the volume being sent either up or down in the SGL machine. Synchronization is a fixed overhead for the barrier-synchronization effect of a gather or scatter operation. Local computation has a local speed parameter c_i measured in Flops/seconds, to be multiplied by w_i the number of local Flops executed. Communication has a coefficient which also varies with the position in the SGL machine but also with the direction of communication, up (gather) or down (scatter). The parameter is measured in seconds/word or seconds/Byte. Sychronization time is also "local" to a node when communicating with its children. It is measured in pure seconds. Like program execution, the cost model is recursive to adapt to the SGL machine shape. Hence the value of $Cost_{child_i}$ is itself an application of the same equation. When the machine has more than two levels, indices become sequences of positive integers denoting paths in the machine tree from the global root to any node.

2.2. Programming model

We now present SGL's syntax and operational semantics. It is a core programming language but which is easy to compile into real language notably the C one.
Values: SGL's values are non-negative integers **Nat**, booleans and arrays (vectors) **Vec** built from them. Vectors of vectors **VecVec** are used as input (resp. output) to scatter (resp. gather) operations. The scatter operation will take a vector of vectors in the master and distribute it to workers/sons. The gather operation will invert this process. $\langle v_1, v_2, \ldots v_\ell \rangle \in$ **VecVec** denotes a vector of vectors. $X \in$ **NatLoc** denotes a scalar variable ("location"), to store numbers. In SGL, $X_{i=pid}$ denotes a *master/children* location; X without index denotes a *master* location. $\overrightarrow{V} \in$ ***VecLoc*** denotes a vector location to store arrays. In the same way, $\overrightarrow{V}_{i=pid}$ denotes a *master/children* location; $\overrightarrow{V}$ without index denotes a *master* location. $\widetilde{W} \in$ ***VVecLoc*** denotes a vectorial location, to store arrays of arrays.
Expressions: they are relatively standard with the convenience of scalar-to-vector (sequential) operations. $\odot$ denotes a binary arithmetic operation. a denotes a scalar arithmetic expression $::= n \mid X \mid a \odot a \mid \overrightarrow{V}[a]$; b denotes a scalar boolean expression; v denotes a vectorial expression $::= \langle a_1, a_2, \ldots a_\ell \rangle \mid \overrightarrow{V} \mid v \odot a \mid v \odot v \mid \widetilde{W}[a]$; w denotes a vectorial-vectorial expression $::= \langle v_1, v_2, ..v_\ell \rangle \mid \widetilde{W}$.
Commands: The language's commands include classical core sequential constructs with SGL's 3 primitives scatter, pardo and gather. c denotes a primitive command. **Com** $::= X := a \mid \overrightarrow{V} := v \mid \widetilde{W} := w \mid c \,;\, c \mid$ **if** b **then** c **else** $c \mid$ **for** X **from** a **to** a **do** c $\mid$ **scatter** w **to** $\overrightarrow{V} \mid$ **scatter** v **to** $X \mid$ **gather** $\overrightarrow{V}$ **to** $\widetilde{W}$ $\mid$ **gather** X **to** $\overrightarrow{V} \mid$ **pardo** $c \mid$ **if master** c **else** c.
Auxiliary functions return structure sizes: **numChd** $\mid$ **len** $\overrightarrow{V} \mid$ **len** $\widetilde{W}$.
States (or environments) σ are maps from imperative variables (locations) to values of the corresponding sort. $\sigma(X) \in Nat$, $\sigma(\overrightarrow{V}) \in Vec$ and $\sigma(\widetilde{W}) \in VecVec$. Here $Pos \in Nat$ is what we call the (relative) *position* of location: $Pos = 0$ denotes *master* position (same as above), and $Pos = i \in \{1..p\}$ denotes position in i-th child. It is the recursive analog of BSP's (or MPI's) `pid`'s. $\sigma(X_{pos}) = \sigma_{pos}(X) \in Nat$, $\sigma(\overrightarrow{V}_{pos}) = \sigma_{pos}(\overrightarrow{V}) \in Vec$. The semantics of SGL primitives is defined by the inference rules in Fig. 1 where a numerator is a hypothesis, a denominator is a conclusion and the arrow denotes execution from a program and initial environ to a final environment. Scattering a vector of vectors (located in the master) to a vector variable $\overrightarrow{V}$ amounts to assigning the individual vectors to that variable's value in the son of the same rank. Scattering a vector to a scalar variable X amounts to assigning the individual elements to that variable's value in the son of the same rank. Gathering a vector variable into a vector of vectors variable is the same as assining the vector variable's value from son i to the vector of vector's value, at rank i in the master. Gathering a scalar variable into a vector variable is the same as assining the scalar variable's value from son i to vector's value, at rank i in the master. The pardo of a sub-program c amounts to executing c is each of the sons.

$$\frac{\langle w,\ \sigma\rangle \to \langle v_1, v_2, \ldots v_\ell\rangle \quad \forall_{i=1}^{numChd} \langle \overrightarrow{V}_i := v_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter}\ w\ \textbf{to}\ \overrightarrow{V},\ \sigma\rangle \to \sigma'}$$

$$\frac{\langle v,\ \sigma\rangle \to \langle n_1, n_2, \ldots n_\ell\rangle \quad \forall_{i=1}^{numChd} \langle X_i := n_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter}\ v\ \textbf{to}\ X,\ \sigma\rangle \to \sigma'}$$

$$\frac{\langle \widetilde{W} := \langle \overrightarrow{V}_1, \overrightarrow{V}_2, \ldots \overrightarrow{V}_{numChd}\rangle,\ \sigma\rangle \to \sigma'}{\langle \textbf{gather}\ \overrightarrow{V}\ \textbf{to}\ \widetilde{W},\ \sigma\rangle \to \sigma'}$$

$$\frac{\langle \overrightarrow{V} := \langle X_1, X_2, \ldots X_{numChd}\rangle,\ \sigma\rangle \to \sigma'}{\langle \textbf{gather}\ X\ \textbf{to}\ \overrightarrow{V},\ \sigma\rangle \to \sigma'}$$

$$\frac{\forall_{i=1}^{numChd} \langle c,\ \sigma_i\rangle \to \sigma'_i}{\langle \textbf{pardo}\ c,\ \sigma\rangle \to \sigma'}$$

Figure 1. Operational semantics of SGL

The `if master` conditional selects an instruction branch depending on the local node's number of children: 0 or more. Also, SGL's communications do not provide direct "horizontal" communications because of their hierarchical *logical* structure. But if the hardware supports them, inter-worker communications can be extracted from SGL semantics either statically or dynamically. To support this claim we now present a result which shows how successive supersteps can be partially compressed into a single horizontal communication-synchronization phase. Simple examples of SGL programs can be found in technical report [14].

SGL's communication primitives are simpler than those of BSP, MPI or most general parallel models. This raises the question of whether the language can really avoid "horizontal" messages. For example domain decomposition methods for PDEs and many data-parallel algorithms are naturally expressed with general communications that do no origin from- or concentrate on a single node. The solution we propose is to allow SGL to express horizontal communications indirectly: a compiler can detect gather-scatter sequences and analyze their effect into a general data exchange among sibling nodes. The full development of this technique is not yet complete but its basis is the formal property below.

Let $\mathcal{G} \equiv$ **gather** $\overrightarrow{V}$ **to** $\widetilde{W}$ and $\mathcal{S} \equiv$ **scatter** $\widetilde{W}$ **to** $\overrightarrow{V}$ in a system with one master and p workers. Assume also that values for $\overrightarrow{V}$ are all vectors of length p. As a result values for $\widetilde{W}$ are equivalent to $p \times p$ matrices of scalars. SGL code for reorganizing such a parallel matrix of values is a sequence $\mathcal{G};\mathcal{P};\mathcal{S}$ where $\mathcal{P}$ is a sequential program in the master that realizes a permutation of the matrix.

Proposition 1. *Let $\mathcal{G};\mathcal{P};\mathcal{S}$ be as above, $\mathcal{P}$ a sequential program whose non-local variables are $\widetilde{W}, \overrightarrow{V}$, and π a permutation of $\{1,\ldots,p\}^2$ such that (π:) $\forall i,j.\ \sigma''(\widetilde{W})_{i,j} = \sigma'(\widetilde{W}_{(\pi(i,j))})$ whenever $\langle \mathcal{P},\ \sigma' \rangle \to \sigma''$. Then $\sigma'''(\overrightarrow{V})_{(i,j)} = \sigma(\overrightarrow{V})_{\pi(i,j)}$ whenever $\langle (\mathcal{G};\mathcal{P};\mathcal{S}),\ \sigma \rangle \to \sigma'''$.*

Proof. The subprograms must evaluate through steps: (g) $\langle \mathcal{G},\ \sigma \rangle \to \sigma'$; (p) $\langle \mathcal{P},\ \sigma' \rangle \to \sigma''$ and (s) $\langle \mathcal{S},\ \sigma'' \rangle \to \sigma'''$. Recall that environments σ are maps from identifiers and machine positions (master, son 1, son 2, son of son i ...) to values. The former are written as indices. The semantics translates step (g) into (g':) $\sigma' = \sigma[\widetilde{W}/\sigma(\overrightarrow{V})_i \mid i = 1,\ldots,p]$ and step (s) into (s':) $\sigma''' = \sigma''[\overrightarrow{V}/\sigma''(\widetilde{W})_i \mid i = 1,\ldots,p]$. We thus have:

$$\begin{aligned} \sigma'''(\overrightarrow{V})_{(i,j)} = (\sigma'''(\overrightarrow{V})_i)_j &= (\sigma''(\widetilde{W})_i)_j && (s') \\ &= \sigma''(\widetilde{W})_{(i,j)} \\ &= \sigma'(\widetilde{W})_{\pi(i,j)} && (\pi) \\ &= \sigma(\overrightarrow{V})_{\pi(i,j)} && (g').\ \Box \end{aligned}$$

If the proposition's hypotheses are satisfied then permutation π can be applied locally to the subset of matrix data available on one worker node, and then given as local argument to a collective communication operation, thus combining two vertical communications into a single horizontal one. The local interpretation of subprogram $\mathcal{P}$ into π is beyond the scope of this paper and requires a target language other than SGL.

3. A SKELETON LIBRARY

3.1. Skeleton paradigm

There exist two kinds of algorithmic skeletons [11]: tasks and data-parallel ones. The former can capture parallelism that originates from executing several tasks, *i.e.* different function calls, in parallel. They mainly describe various patterns for organizing parallelism, including pipelining, farming, client-server, *etc.* The latter parallelize computation on a data structure by partitioning it among processors and performing computation simultaneously on different parts of it.

A well-know disadvantage of skeleton languages is that the only admitted parallelism is usually that of skeletons, while many parallel applications are not easily expressible as instances of known skeletons. Skeleton languages must be constructed as to allow the integration of skeletal and ad-hoc parallelism in a well defined way [4]. In this light, having skeletons in SGL would combine the expressive power of collective communication patterns with the clarity of the skeleton approach.

In this work we consider the implementation of well-known data-parallel skeletons because they are simpler to use than task-parallel ones for coarse-grained models[2] and also because they encode many scientific computation problems and scale naturally. Even if the SGL's implementation is certainly less efficient compared to a dedicated skeleton language (using MPI send/receive [6]), the programmer can compose skeletons when it is natural for him and use a SGL programming style when new patterns are needed.

3.2. Our set of data-parallel skeletons

Fig. 2 defines the functional semantics of a set of data-parallel skeletons [4, 1]. It can also be seen as a naive

[2]An efficient BSP implementation for those has nevertheless been shown in [7]

$$
\begin{aligned}
\mathbf{repl}\, x\, n &= [x, \ldots, x] \\
\mathbf{map}\ f\ [x_1, \ldots, x_n] &= [(f\, x_1), \ldots, (f\, x_n)] \\
\mathbf{mapidx}\ g\ [x_1, \ldots, x_n] &= [(g\, 1\, x_1), \ldots, (g\, n\, x_n)] \\
\mathbf{zip}\ \oplus\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] &= [x_1 \oplus y_1, \ldots, x_n \oplus y_n] \\
\mathbf{reduce}\ \oplus\ [x_1, \ldots, x_n] &= x_1 \oplus \cdots \oplus x_n \\
\mathbf{scan}\ \oplus\ [x_1, \ldots, x_n] &= [x_1, (x_1 \oplus x_2), \cdots, \\
&\qquad ((x_1 \oplus x_2) \cdots \oplus x_n)]
\end{aligned}
$$

Figure 2. Simple data-parallel skeletons

sequential implementation using lists. The skeletons work as follow. Skeleton **repl** creates a new list containing n times element x. Here we speak of lists for the specification but parallel implementations would use more efficient data-structures as arrays (in this article) or a stream (in a client/server or grid environment) since the size of the lists remain constant.

The **map**, **mapidx** and **zip** skeletons are equivalent to the classical *Single-Program-Multiple-Data* (SPMD) style of parallel programming, where a single program f is applied to different data in parallel. Parallel execution is obtained by assigning a share of the input list to each available processor.

reduce is an elementary data-parallel skeleton: the function **reduce** $\oplus\ e\, l$ computes the "sum" of all elements in a list l, using the associative binary operator $\oplus$ and a neutral element e. Reduction has traditionally been very popular in parallel programming and is provided as the collective operation `MPI_Reduce` in the MPI standard. Note that the binary operator $\oplus$ may itself be time-consuming. To parallelize the **reduce** skeleton, the input list is divided into sub-lists that are assigned to each processor. The processors compute the $\oplus$-reductions of their elements locally in parallel, and the local results are then combined either on a single processor or using a tree-like pattern of computation and communication, making use of associativity in the binary operator.

The **scan** skeleton is similar to **reduce** (and is provided as the collective operation `MPI_Scan`), but rather than the single "sum" produced by **reduce**, **scan** computes the partial (prefix) sums for all list elements. Parallel implementation is done as for **reduce**.

reduce and **scan** are basic skeletons that are like the MPI's collective operations. A more complex data-parallel skeleton, the Distributable Homomorphism **dh** presented in [1], is used to express a special class of divide-and-conquer algorithms. $dh\ \oplus \otimes l$ transforms a list $l = [x_1, \cdots, x_n]$ of size $n = 2^m$ into a result list $r = [y_1, \cdots, y_n]$ of the same size, whose elements are recursively computed as follows:

$$
y_i = \begin{cases} u_i \oplus v_i & \text{if } i \leq \frac{n}{2} \\ u_{i-\frac{n}{2}} \otimes v_{i-\frac{n}{2}} & \text{otherwise} \end{cases}
$$

where $u = \mathbf{dh}\ \oplus\ \times [x_1, \ldots, x_{\frac{n}{2}}]$, *i.e.* **dh** applied to the left half of the input list l and $v = \mathbf{dh}\ \oplus\ \times [x_{\frac{n}{2}+1}, \ldots, x_n]$, *i.e.* **dh** applied to the right half of l. The **dh** skeleton provides the well-known butterfly pattern of computation which can be used to implement many computations.

4. PROGRAM EXAMPLES

In this section, we give some example of classical parallel numerical computations that can be performed using the skeletons presented above.

4.1. Tridiagonal System Solver (TDS)

As first application example, we consider the solution of the TDS of equations [1]: $A.x = b$ where A is a $n \times n$ sparse matrix representing coefficients, x a vector of unknowns and b a right-hand-side vector. The only values of matrix A different from 0 are on the main diagonal, as well as directly above and below it — we call them the upper- and lower diagonal, respectively.

We represent the TDS as a list of rows, each inner row consisting of four values (a_1, a_2, a_3, a_4): the value a_1 that is part of the lower diagonal of matrix A, the value a_2 at the main diagonal, the value a_3 at the upper diagonal, and the value a_4 that is part of the right-hand-side vector b. The first and last row of A only contain two values, but in order to obtain a consistent representation we set $a_1 = 0$ for the first and $a_3 = 0$ for the last row. This corresponds to adding a column of zeroes at the left and right of matrix A.

We now use the **dh** skeleton to parallelize the problem as a divide-and-conquer parallel algorithm. Since in the conquer phase, two subsystems can be combined using the first and last row of the systems, our implementation works on triples (a, f, l) of rows, containing for each initial input row the actual row value a, and the first f and the last l row of the subsystem the row is part of. Using this list representation, the algorithm can be expressed as follows:
$(tds\, m) = \mathbf{map}\, \pi_1\, (\mathbf{dh}\ \oplus \otimes (\mathbf{map}\, triple\, m))$ where:

$$
\begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \oplus \begin{pmatrix} a_2 \\ f_2 \\ l_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} a_1 \diamond (l_1 \star f_2) \\ f_1 \diamond (l_1 \star f_2) \\ (l_1 \circ f_2) \bullet l_2 \end{pmatrix}
$$

$$
\begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \otimes \begin{pmatrix} a_2 \\ f_2 \\ l_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} (l_1 \circ f_2) \bullet a_2 \\ (f_1 \diamond (l_1 \star f_2) \\ (l_1 \circ f_2) \bullet l_2 \end{pmatrix}
$$

and where (if $a = (a_1, a_2, a_3, a_4)$ and $b = (b_1, b_2, b_3, b_4)$):

$$\begin{array}{rcl} a \star b & = & (a_1, a_3 - (\frac{a_2}{b_1}) \times b_2, b_3 \times (-\frac{a_2}{b_1}), a_4 - (\frac{a_2}{b_1}) \times b_4) \\ a \diamond b & = & (a_1 - (\frac{a_3}{b_2}) \times b_1, a_2, (-\frac{a_3}{b_2}) \times b_3, a_4 - (\frac{a_3}{b_2}) \times b_4) \\ a \circ b & = & (a_1, a_2 - (b_1 \times \frac{a_3}{b_2}), (-b_3 \times a_3/b_2), a_4 - (\frac{a_3}{b_2}) \times b_4) \\ a \bullet b & = & (a_1, -(\frac{a_2}{b_1}) \times b_2, a_3 - (b_3 \times \frac{a_2}{b_1}), a_4 - \frac{b_4 \times a_2}{b_1}) \end{array}$$

which are row operations in a Gaussian elimination.

The **dh** method works as follow. In the divide phase, the matrix is subdivided into single rows. The conquer phase starts by combining neighbouring rows, applying first $\star$ (resp. $\circ$) and then $\diamond$ (resp. $\bullet$), which results in systems of two equations each, with non-zero elements in the first column, the main diagonal and the last column. The sub-matrices are then combined into matrices of four rows with the same structure, *i.e.*, where all non-zero elements are either on the diagonal, or in the first or last column. This process continues until, finally, the entire system of equations has this form. Note that the first and last column of the matrix remain zero throughout the process, thus the the solution for the initial system of equations.

Combining two subsystems is achieved using a special row, obtained from the last row l of the first system and the first row f of the second one, using operator $\star$ (resp. $\circ$). This row is applied using operator $\diamond$ (resp. $\bullet$) to each row of the first system. Similarly, the rows of the second subsystem are adjusted using another special row obtained from the first and last rows.

4.2. Fast Fourier Transform (FFT)

The FFT of a list $x = [x_0, \ldots, x_{n-1}]$ of length $n = 2^m$ yields a list whose ith element is defined as:

$$(FFT\, x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$$

where ω_n denotes the nth complex root of unity $e^{2\pi\sqrt{-1}/n}$.

The FFT can be expressed in a divide-and-conquer form:

$$(FFT\, x)_i = \begin{cases} (FFT\, u)_i \oplus_{i,n} (FFT\, v)_i & \text{if } i < \frac{n}{2} \\ (FFT\, u)_j \otimes_{j,n} (FFT\, v)_j & \text{otherwise} \end{cases}$$

where $u = [x_0, x_2, \ldots, x_{n-2}]$, $v = [x_1, x_3, \ldots, x_{n-1}]$, $j = i - \frac{n}{2}$, and $a \oplus_{i,n} b = a + \omega_n^i b$ and $a \otimes_{j,n} b = a - \omega_n^j b$. This formulation is close to the **dh** skeleton except $\oplus_{i,n}$ and $\otimes_{j,n}$ being parametrized with i and n.

These operators repeatedly compute the roots of unity. Instead of computing them for every call, they can be computed once *a priori* and stored in a list $\Omega = [\omega_n^1, \ldots, \omega_n^{\frac{n}{2}}]$ accessible by both operators. For this, we first use a **scan**. FFT can thus be expressed as follow:

$$\begin{array}{rcl} (FFT\, l) & = & \textbf{let}\, \Omega = \textbf{scan} + 1\, (\textbf{repl}\, (\omega\, n)\, \frac{n}{2}) \\ & & \textbf{in map}\, \pi_1\, (\textbf{dh} \oplus \otimes (\textbf{mapidx}\, \mathit{triple}\, l)) \end{array}$$

where:

$$\begin{pmatrix} x_1 \\ i_1 \\ n_1 \end{pmatrix} \oplus \begin{pmatrix} x_2 \\ i_2 \\ n_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} x_1 \oplus_{n_1}^{i_1} x_2 \\ i_1 \\ 2n_1 \end{pmatrix}$$

($\otimes$ is defined similarly). The first element of each triple contains the input value, the second one its position and the last one the current list length. In each **dh** step, these operators are applied element-wise to two lists of length $n_1 = n_2$, resulting in a list of length $2n_1$. $x_1 \oplus_{n_1}^{i_1} x_2$ (resp. for $\otimes$) is defined as $x_1 + x_2 \times (th\, (n \times \frac{i_1}{n_2})\, \Omega)$ where $th\, n\, [x_1, \ldots, x_n, \ldots] = x_n$.

5. IMPLEMENTATION

We have implemented the **dh** skeleton using SGL which map its model of execution since they are both recursive ones. We have then apply it to TDS and FFT. For each algorithm we compare the model's predicted vs observed run time for increasing data size and variating the number of processors and cores. Finally we computed speed-up values.

5.1. Implementation of dh using SGL

All data is distributed in p *workers* and the algorithm performs recursively as follow: first of all, each *worker* performs a sequential **dh** with its own local data; then, the *master* gathers the computed data, permutes them according the position, and scatters the permuted data to the *workers*; after that, each *worker* performs either $\oplus$ operation or $\otimes$ operation according its position. After $\log(p)$ times above achievements, we obtain the final result.

In the pseudo code, line 3 is a recursive call to the algorithms, lines 14 - 18 are executed in parallel, and lines 22 - 31, the no-children case, represent a local sequential loop. The cost of the super-step is below:

$$\begin{array}{rcl} Cost_{Master} & = & \max_{i=1..p}(DH_{Child_i}) + \\ & & \log(p) \times (2^n \times (g_\uparrow + g_\downarrow) + 2l \\ & & +2^n \times \max(c_\oplus, c_\otimes)) \end{array}$$

$$Cost_{Worker} = 2^n \times n \times \frac{c_\oplus + c_\otimes}{2}$$

dh imposes many "horizontal" communications that would be programmed in SGL as sequences of the form $\mathcal{G}; \mathcal{P}; \mathcal{S}$ as in the proposition. A SGL compiler or even interpreter will be able to optimize them into a flat horizontal communication-synchronization phase, and this is how we have implemented it in our experiments. Also, it is easy to see, using our

Algorithm 1 Distributable Homomorphism (dh)

```
DH(IN $\oplus$, IN $\otimes$, INOUT $data$)
if master then
    par do
        DH($\oplus$, $\otimes$, $data$);
    end par
    for n from 1 to log2(numChd) do
        gather $data$ to $tmp$;
        for i from 1 to (len(numChd)) do
            if ((i-1) % exp2(n))/2 = 0 then
                Swap($tmp$[i], $tmp$[i+exp2(n)])
            end if
        end for
        scatter $tmp$ to $list$;
        par do
            if ((PID-1) % exp2(n))/2 = 0 then
                data := data $\oplus$ list;
            else
                date := data $\otimes$ list;
            end if
        end par
    end for
else
    for n from 1 to log2(len($data$)) do
        for i from 1 to (len($data$)) do
            if ((i-1) % exp2(n))/2 = 0 then
                $tmp$ := $data$[i] $\oplus$ $data$[i+exp2(n)];
            else
                $tmp$ := $data$[i-exp2(n)] $\otimes$ $data$[i];
            end if
        end for
        $data$ := $tmp$;
    end for
end if
```

semantics, that the SGL's implementation of **dh** is a correct one thanks to the simple programming model.

5.2. Performance Measurement

The local processing speed c and network parameters l, g can be measured on a chosen machine once the hierarchy of processors and benchmarking setup has been defined. For a given algorithm we can also analyze theoretically, estimate or count the w quantities. We have performed experiments to observe how the measured computation time of an SGL algorithm compares with the cost model's prediction. The overall system has the following description: The machine is a SGI Altix ICE 8200EX with 32 computing nodes, each node has 2 Intel Xeon E5440 (Quad-core, 2.83 GHz, 1333 FSB) CPUs; 1 TB in total (4 GB per core, DDR2-667 1.5ns) and the network is 4X DDR InfiniBand switches (16 Gbit/s). We build a 2-level SGL abstract machine to represent a part of this physical machine:

Unit	Children	Communication
Root-master	8 nodes	InfiniBand
Node-master	4 cores	Front-Side Bus
Worker	0	N/A

The CPUs of computing nodes are clocked at 2.83 GHz, for each one $c = 0.000353\mu s/op$.

For node level, we use the collective functions *MPI_Barrier* for L, *MPI_Scatterv* for $g_{\downarrow}$, and *MPI_Gatherv* for $g_{\uparrow}$ of SGI's Message Passing Toolkit (MPT 2.04). The g values are given in $\mu s/32bits$.

Machine	NbProc	L(μs)	$g_{\downarrow}$	$g_{\uparrow}$
2nodes x 1core	2	1.48	0.00138	0.00215
4nodes x 1core	4	2.85	0.00169	0.00200
8nodes x 1core	8	4.37	0.00189	0.00205
16nodes x 1core	16	5.96	0.00204	0.00209

At the core (worker) level, we use OpenMP's Barrier for L and the C language's function *memcpy* for g:

Machine	L (μs)	g ($\mu s/32bits$)
2 cores	12.08	0.00059
4 cores	25.64	0.00059

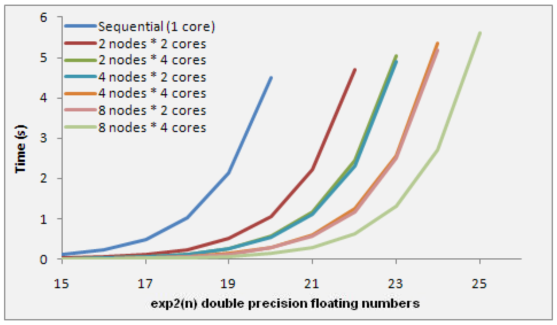

Figure 3. TDS

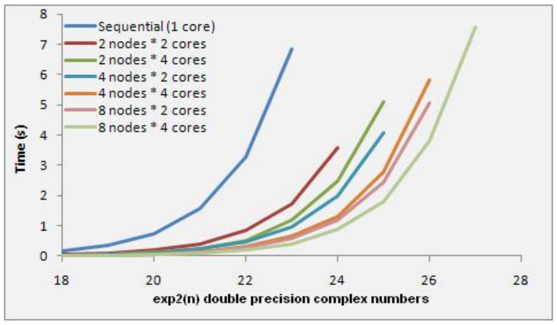

Figure 4. FFT

6. RELATED WORKS

[4] described how to add skeletons in MPI (the eSkel library) as well as some experiments. It also gives convincing and pragmatic arguments for the use of

mixed message passing and skeleton programming using C. Implementation of skeletons using a BSP library was first done in [19] and BSP cost prediction for skeletons has also been done in [9]. In both works, a C library was used and the set of skeletons is more restricted than ours.

Performing hybrid computation with both MPI and OpenMP is not new and is known to be efficient [17]. OpenMP is also known to be more efficient on multi-processor multi-cores architectures than MPI [15] even if the difference is currently smaller than what many researcher think it is. Combining both produces naturally yields efficient parallel programs. Except in [10, 3], we are not aware of an implementation of skeletons using both MPI and OpenMP. In [16], the authors present a GPGPU implementation of their skeleton library. Our implementation is more generic since it can run on a hierarchical parallel model. When having an implementation of our operators for GPGPU architectures, we will have this implementation for our set of skeletons.

MapReduce [5] is a well known and simple-to-use framework to support distributed computing on large data sets. But it is not as expressive as our set of skeletons. One reason for this is that our semantics contains data layout on the hierarchical architecture so an SGL program can write explicite data movements: SGL works on a lower level than MapReduce but it provides a good performance prediction and optimization thanks for its cost model and the methods described here allow both more detailed (SGL) and more abstract (skeletons) programming than MapReduce.

7. CONCLUSION

In this paper, we have implemented some data-parallel skeletons using a model called SGL well-adapted for hierarchical and hybrid architectures. This work is inspired by [8]. We have also performed some benchmarks for the skeletons expressed in SGL. SGL is of interest for its cost model, allowing an estimation of the execution time of its programs. We have defined the SGL cost of each skeleton's implementation. We have then compared predicted performances with measured ones of some numerical applications of our set of skeletons. We have show good performance (scalability) and good predictions (predictability).

We are currently working to modify our implementation in a similar manner to the SkeTo library (`http://www.ipl.t.u-tokyo.ac.jp/sketo/`): that library provides data-parallel skeletons for C++/MPI programmers where skeleton's expressions are template expressions. This provide efficiency and fast development even for the relatively large applications considered by the authors of SkeTo. We are also working on the automatic optimization for the horizontal communications with the proposition/proof in section 2.2 to automate certain optimizations that were done manually.

Task-parallel skeletons are also useful for many applications and can be combined with data-parallel skeletons. It would be interesting to include them in our set of skeletons. Furthermore, it is theoretically possible to have nested skeletons *e.g.* by replacing the f argument of the **map** by any parallel function or skeleton. Currently, our implementation forbids this. Flattening such nested parallelism is theoretically possible but no formal tool exists for this. The definition and development of all the above tools is an important challenge.

REFERENCES

[1] M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance.* PhD thesis, Universität Münster, 2007.

[2] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

[3] P. Ciechanowicz and H. Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *HPCC*, pages 108–113. IEEE, 2010.

[4] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[5] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[6] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste. QUAFF : Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[7] I. Garnier and F. Gava. CPS Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons. *Parallel, Emergent and Distributed Systems*, 2011. To appear.

[8] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004.

[9] Y. Hayashi and M. Cole. Automated BSP cost analysis of a parallel maximum segment sum program derivation. *Parallel Processing Letters*, 12(1):95–112, 2002.

[10] Y. Karasawa and H. Iwasaki. A parallel skeleton library for multi-core clusters. In *ICPP*, pages 84–91. IEEE Computer Society, 2009.

[11] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002.

[12] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[13] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

[14] Chong Li and G. Hains. A simple bridging model for high-performance computing. Technical Report TR-LACL-2010-12, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (UPEC-Paris 12), 2010.

[15] D.A. Mallon, G.L. Taboada, C. Teijeiro, J. Tourino, B.B. Fraguela, A. Gomez, R. Doallo, and J.C. Mourino. Performance evaluation of MPI, UPC and OpenMP on multicore architectures. In *Euro PVM/MPI*, LNCS, pages 174–184. Springer, 2009.

[16] S. Sato and H. Iwasaki. A skeletal parallel framework with fusion optimizer for GPGPU programming. In *APLAS*, volume 5904 of *LNCS*, pages 79–94. Springer, 2009.

[17] L. Smith and M. Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9:83–89, 2001.

[18] L. G. Valiant. A bridging model for multi-core computing. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] A. Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393–407, 2001.