SGL Hierarchical Bridging Model for Parallel Programming

LI Chong[§†] HAINS Gaétan[†‡]

Parallel programming and data-parallel algorithms have been the main techniques supporting high-performance computing for many decades. Like all non-functional properties of software, the conversion of computing resources into scalable and predictable performance involves a delicate balance of abstraction and automation with semantic precision. A major conceptual step was taken by L. Valiant who introduced the Bulk-Synchronous Parallel (BSP) model. Parallel algorithms on BSP can be designed and measured by taking into account not only the classical balance between time and parallel space but also communication and synchronization. However, the flat view of a parallel machine as a set of communicating sequential machines remains true but is more and more incomplete. Inspired by BSP, we introduce here the SGL bridging model, aim to improve the simplicity of parallel program development, the portability of parallel program code, and the precision of parallel algorithm performance prediction on both classical parallel machines and novel hierarchical machines. SGL can be used later as a part of model transformation for generating automatically efficient parallel program with its cost model.

1 Introduction

Parallel programming and data-parallel algorithms have been the main techniques supporting high-performance computing for many decades. A major conceptual step was taken by L. Valiant who introduced the Bulk-Synchronous Parallel (BSP) model [23]. Inspired by the complexity theory of PRAM model of parallel computers [6], Valiant proposed that parallel algorithms can be designed and measured by taking into account not only the classical balance between time and parallel space (hence the number of processors) but also communication and synchronization. The BSP performance model is both realistic and tractable so that researchers like McColl et al. [19] were able to define BSP versions of all important PRAM algorithms, implement them and verify their portable and scalable performances as predicted by the model. BSP is thus a *bridging model* relating parallel algorithms to hardware architectures.

From the mid-1990's, it became clear that BSP is the model of choice for implementing algorithmic skeletons: its view of the parallel system included explicit processes and added a small set of network performance parameters to allow predictable performance. Hains et al. designed BS-lambda [18] as a minimal model of computation with BSP operations. BS-lambda became the basis for Bulk-Synchronous ML (BSML) [17] a variant of CAML under development by Loulergue et al. since 2000. BSML has a simplified programming interface of only four operations: `mkpar` to construct parallel vectors indexed by processors, `proj` to map them back to lists/arrays, `apply` to generate asynchronous parallel computation and `put` to generate communication and global synchronization from a two-dimensional processor-processor pairing. As a result, parallel performance mathematically follows

§, 国立情報学研究所, National Institute of Informatics.
†, パリ大学（東）, Université Paris-Est, France.
‡, 国立応用科学院サントル校, INSA Centre, France.

from program semantics and the BSP parameters of the host architecture.

While BSML was evolving and practical experience with BSP algorithms was accumulating, one of its basic assumptions about parallel hardware was changing. The flat view of a parallel machine as a set of communicating sequential machines remains true but is more and more incomplete. Recent supercomputers like IBM Blue Gene [21] features multi-processors on one card, multi-core processors on one chip, multiple-rack clusters etc. The Cell/B.E. [9], Cell-based supercomputer RoadRunner [2] and GPU-equipped hybrid machines feature a CPU with Master-Worker architecture. Moreover, [11] observes that heterogeneous chip multiprocessors present unique opportunities for improving system throughput and reducing processor consumption. The trend towards *Green Computing* puts even more pressure on the optimal use of architectures that are not only highly scalable but hierarchical and non-homogeneous.

Li and Hains [14] [15] proposed a hierarchical bridging model named SGL for modelling heterogeneous parallel architectures and providing a programming platform for parallel algorithms development. SGL attempts to improve the simplicity of parallel program development, the portability of parallel program code, and the precision of parallel algorithm performance prediction on both classical parallel machines and novel hierarchical machines.

Several skeletons such as parallel scan, parallel sort, distributed homomorphism, etc. have been implemented on SGL [15] [13] to attempt to motivate and support the view that BSP's advantages for parallel software can be enhanced by the recursive hierarchical and heterogeneous machine structure of SGL, while simplifying the programming interface even further by replacing point-to-point messages with logically centralised communications.

The communication on SGL is conceptually centralized. SGL does not express "horizontal" communication patterns, even this defect affects a minority of algorithms and can be compensated by automated compilation/interpretation. Therefore, the GPS theorem was proposed in [16] that can be implemented later in a compiler to optimize the SGL's "horizontal" all-to-all communication.

The programming model of SGL replaces the BSML programming primitives with *scatter*, *gather* and *pardo*. The SGL cost model improves the clarity of algorithms performance analysis. At the same time, it allows benchmarking machine performance and algorithm scalability. SGL provides the possibility to divide parallel programming into three different dimensions in order to reduce the development complexity. Then the tasks on each dimension can be conquered independently by either system engineer, parallelism specialist, or end-use domain expert.

The rest of this paper is organized as follow: Section 2 gives an overview of the SGL model; Section 3 presents skeletal parallelism on SGL; Section 4 introduces the GPS theorem for horizontal communication on SGL; and Section 5 is the conclusion.

2 The SGL Model

2.1 The SGL Computer

The PRAM model [6] of parallel computation begins with a set of p sequential Von-Neumann machines (*processors*). It then connects them with a shared memory and an implicit global controller (as in a SIMD architecture). The BSP model [23] relaxes the global control and direct memory access hypotheses. It also begins with a set of p sequential machines but assumes asynchronous execution except for global synchronisation barriers and point-to-point communications that complete between two successive barriers. It is common to

assume that the PRAM/BSP processors are identical, but it is relatively easy to adapt algorithms to a set of heterogeneous processors with varied processing speeds. In both cases the set of processors has no structure other than its numbering from 0 to $p-1$. This absence of structure on the set of processors can be traced back to the failure of a trend of popular research in the 1980s: algorithms for specific interconnect topologies such as hypercube algorithms, systolic algorithms, rectangular grid algorithms, etc. We trace this failure to the excessive variety of algorithms and architectures without a model to *bridge* the portability gap between algorithms and parallel machines.

The SGL model [15] we proposed does introduce a degree of topology on the set of processors. But this topology is purely hierarchical and is not intended to become an explicit factor of algorithm diversity. Just as PRAM/BSP algorithms use the p parameter to adapt to systems of all sizes, our SGL algorithms use the machine parameters to adapt to all possible systems.

The abstract machine or an SGL Computer (SGLC) is defined as the combination of three attributes (図 1):

1. A set of **workers**, which are sequential processors composed of a computation element ("computing core") and local memory unit. The *workers* provide the primary computing power.
2. A set of **masters**, which are also sequential processors composed of a coordination element ("coordinating core") and central memory unit whose data is accessible by its own children via the coordinating core, i.e. the coordinating core can *scatter* (resp. *gather*) message to (resp. from) its children. The *masters* limit the communication cost.
3. A **tree** structure that the root is a *master* and the *master*'s children may be either *masters* themselves or leaf-*workers*. The number of children is unbounded so that the BSP/PRAM concept of a flat p-vector of processors is easily simulated in SGL. The *tree* structure offers a vertical scalability to SGLC that the BSP concept can scale only horizontally the number of processors.

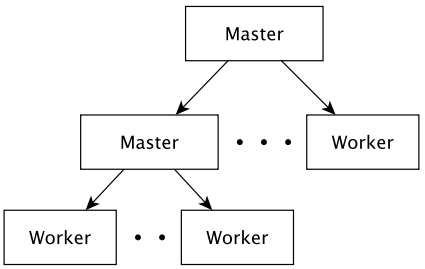

図 1 A multi-level SGL computer

2.2 SGL Execution Model

An SGL *program execution* is a sequence of *supersteps*. The initial computing data and final result can be either distributed in workers or centralised in the *root-master*. Each *superstep* is composed of four phases:

1. A **scatter communication** phase initiated by the *master*. The *master* scatters data to its *children* if the data is not yet distributed, then it engages its *children* to start the second phase.
2. An **asynchronous computation** phase performed by the *children*. The *children* execute the task initialized by their *master* in the first phase. A *child* can be either a *master* or a *worker*. The *workers* accomplish its *master*'s task; and the *child-master* can start a sequence of *supersteps* with its own children nested in this *child* computation phase so-called *sub-supersteps*.
3. A **gather communication** phase centred on the *master*. The *master* synchronizes its *chil-*

dren's task of the second phase, then gathers the computation results from its *children* if necessary.

4. A **local computation** phase on the *master*. The *master* post-processes the gathered data of the third phase and terminates the *superstep*.

With these four phases, we can usually write an SGL program recursively. The choice of using (or not) children in the recursive program depends on performance parameters that combine (known) parameters of: communication costs, synchronization costs, computation costs, and load balancing.

2.3 SGL Cost Model

Like all bridging models, SGL's main goal of expressing parallel algorithms requires a precise notion of the execution time for a program. We give here the mathematical form of this *cost model* with the understanding that a full definition should be based on the operational semantics and will be defined in a further document. The cost equations below are nevertheless sufficient for informal algorithm design/understanding and comparison with other parallel models.

The cost of an SGL algorithm (i.e. of its execution on given input data) is the sum of all its supersteps' cost. The cost of an individual superstep is composed of computation cost and communication cost. Computation cost is the sum of local computation time in the children (parallel, hence combined with max) and of the local computation in the master. Addition of both terms realizes the hypothesis that the master's local work may not overlap with the children's work. Communication cost is split into the time for performing the scatter and the gather operation. Communication costs are estimated by a linear term similar to BSP's $g \times h + L$, based on machine parameters for (the inverse of) bandwidth and synchronization between the master and its children. As always, local computation cost is the number of instructions executed divided by processing speed.

The machine-dependent parameters can be made as abstract or concrete as desired. In other words, theoretical SGL algorithms can be investigated with respect to their asymptotic complexity classes, while portable concrete SGL algorithms can be analysed for more precise cost formulae using bytecode-like instruction counts and normalized communication parameters, and finally SGL programs can be compiled and measured for actual performance on a given architecture.

Concrete cost estimations and measurements use the following machine and algorithm parameters. Each level of a hierarchy can have different parameter values.

- Machine parameters:
  - **p** : the number of *children* processors that a *master* has. We use P for the total number of leaf-*workers* of a machine.
  - **c** : computation speed of processors, c_0 denotes the time interval for performing a unit of *work* on the *master* and $c_i (i{=}1..p)$ denotes the time interval for performing a unit of *work* on *children* processor. Parameter c without an index refers to a local quantity.
  - **g** : the gap, $g_\downarrow$ defined as the minimum time interval for transmitting one word from *master* to its *children*, and $g_\uparrow$ for *children* to its *master*. We use a single g in case of symmetric communication cost.
  - **l** : the latency to perform a *gather* communication synchronization. i.e. the time to execute a 1-bit gather. Since a *scatter* communication should be initialized by the *master*, synchronisation is not necessary for *scatter* phase. We use L for the total cost of all-level synchronisations of

one global superstep.

- Algorithm parameters:
  - **w** : work or number of local operations performed by processors, w_0 denotes the *master*'s work and $w_i(i{=}1..p)$ denote the work of a *child* i. A parameter w without an index refers to a local quantity.
  - **k** : number of words to transmit, $k_\downarrow$ denotes number of words that *master* scatter to its *children*, and $k_\uparrow$ denotes number of words that *master* gathers from its *children*.

Thus, the cost formula is:

$$Cost_{node} = \quad w \times c + \left[\max_{i=1..p} \{Cost_{child_i}\} + k_\downarrow \times g_\downarrow + k_\uparrow \times g_\uparrow + l \right]$$

which clearly covers the possibility of a heterogeneous architecture.

2.4 SGL Programming Model

Bougé advocated in his paper [3], that an abstract computing model needs an execution model, but also a programming model. We enrich Winskel's basic imperative language IMP [24] to yield our deterministic parallel programming language – SGL.

Values are integers, booleans and arrays (vectors) built from them. Vectors of vectors are necessary for building blocks of work to be scattered among workers. The scatter operation takes a vector of vectors in the master and distributes it to workers/children. The gather operation inverts this process.

Imperative variables are abstractions of memory positions and are called *locations*. They are many-sorted like the language's values.

- X ranges over scalar locations **NatLoc**, i.e. names of memory elements to store numbers. Here $X_{i=pid}$ denotes *master*/*children* locations; X without index denotes *master* location.
- $\overrightarrow{V}$ ranges over vectorial locations **VecLoc**, i.e. names of memory elements to store arrays. Here $\overrightarrow{V}_{i=pid}$ denotes *master*/*children* locations; $\overrightarrow{V}$ without index denotes *master* location.
- $\widetilde{W}$ ranges over vectorial vectorial locations **VVecLoc**, i.e. names of memory elements to store arrays of arrays.

Expressions are relatively standard with the convenience of scalar-to-vector (sequential) operations. $\odot$ ranges over binary arithmetic operations. a ranges over scalar arithmetic expressions::= $n \mid X \mid a \odot a \mid \overrightarrow{V}[a]$ b ranges over scalar boolean expressions. v ranges over vectorial expressions ::= $< a_1, a_2, \ldots a_\ell > \mid \overrightarrow{V} \mid v \odot a \mid v \odot v \mid \widetilde{W}[a]$ and w ranges over vectorial vectorial expressions::= $< v_1, v_2, ..v_\ell > \mid \widetilde{W}$.

The language's commands include classical sequential constructs with SGL's three primitives: **scatter**, **pardo** and **gather**. Their exact meanings of parallel statements are defined in the semantic rules below.

- c ranges over primitive commands.

  Com ::=

  skip | $X := a$ | $\overrightarrow{V} := v$ | $\widetilde{W} := w$ | c ; c
  | **if** b **then** c **else** c
  | **for** X **from** a **to** a **do** c
  | **scatter** w **to** $\overrightarrow{V}$ | **scatter** v **to** X
  | **gather** $\overrightarrow{V}$ **to** $\widetilde{W}$ | **gather** X **to** $\overrightarrow{V}$
  | **pardo** c | **if master then** c **else** c
- Auxiliary commands are also needed.

  Aux ::= **numChd** | **len** $\overrightarrow{V}$ | **len** $\widetilde{W}$

States (or environments) are maps from imperative variables (locations) to values of the corresponding sort. Like values they are many-sorted and we use the following notations and definitions for them. The functions in States Σ are defined as follow:

- $\sigma : NatLoc \to Nat$, thus $\sigma(X) \in Nat$

- $\sigma : VecLoc \to Vec$, thus $\sigma(\overrightarrow{V}) \in Vec$
- $\sigma : VVecLoc \to VecVec$, thus $\sigma(\widetilde{W}) \in VecVec$

Here $Pos \in Nat$ is what we call the (relative) *position*:
$Pos = 0$ 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) `pids`.

- $\sigma : NatLoc \to Pos \to Nat$
- $\sigma : VecLoc \to Pos \to Vec$

The semantics of vector expressions is standard and deserves no special explanations. The evaluation of parallel primitive commands are defined as follow:

- **Scatter**

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

The vector expression v is evaluated in the *master*'s local environment which is a part of the initial global environment σ to get the scalar value of each element of v; and the length of v shall be the same as the number of *children* processors that the *master* has. After that, the scalar value of each element of v is sent to a *child*'s environment that the position of the child is the same as the indexation of the sending element of v. We thus have a new global environment σ' composed of all new local environments of the master and its children. The communication cost of this statement is (p words $\times\ g_\downarrow$).

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

Same as the statement (**scatter** v **to** X) but for scattering vectorial vectorial expression. The communication cost of this statement is ($\sum$ number of words of $v \times g_\downarrow$).

- **Gather**

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

The values in location X on the *children* are sent to the *master*, then they are stored as a vector in the location $\overrightarrow{V}$ on the *master*. A synchronisation is presented here to ensure the reception of all values. The communication cost of this statement is (p words $\times\ g_\uparrow + l$).

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

Same as the statement (**gather** X **to** $\overrightarrow{V}$) but for gathering vectorial vectorial expression. The communication cost of this statement is ($\sum$ number of words of $v \times g_\uparrow + l$).

- **Parallel**

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

Each *child* of the *master* evaluates independently the command c in its own local environment σ_i where the position $i = 1 \ldots p$ and p is the number of children. This statement costs $\max_{i=1..p}\{\text{Cost}(c)_i\}$.

$$\frac{\langle numChd = 0,\ \sigma\rangle \to \textbf{true} \quad \langle c_2, \sigma\rangle \to \sigma'}{\langle \textbf{if master then } c_1 \textbf{ else } c_2,\ \sigma\rangle \to \sigma'}$$

$$\frac{\langle numChd = 0,\ \sigma\rangle \to \textbf{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if master then } c_1 \textbf{ else } c_2,\ \sigma\rangle \to \sigma'}$$

If the local environment is on a *master*, i.e. the number of children is zero in the local environment, the command c_1 will be evaluated; otherwise, the command c_2 will be evaluated.

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

表 1 Simple data-parallel skeletons

3 Skeletal Parallelism

3.1 Skeleton paradigm

There exist two kinds of algorithmic skeletons [10]: 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 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 [5]), the programmer can compose skeletons when it is natural for him and use a SGL programming style when new patterns are needed.

3.2 Data-parallel skeletons

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

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

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.

3.3 Program Example

We give here an example of classical parallel numerical computation that can be performed using the skeletons presented above.

The Fast Fourier Transformation (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:

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

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:

$$\begin{aligned} (FFT\, l) \quad = \quad & \mathbf{let}\, \Omega = \mathbf{scan}\ +\ 1\, (\mathbf{repl}\, (\omega\, n)\, \tfrac{n}{2}) \\ & \mathbf{in\ map}\, \pi_1\, (\mathbf{dh} \oplus \otimes\, (\mathbf{mapidx}\, triple\, l)) \end{aligned}$$

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 elementwise 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 $th\, n\, [x_1, \ldots, x_n, \ldots] = x_n$.

3.4 Implementation of DH on SGL

The implementation of basic skeletons on SGL can be found in [12]. Here we show the implementation of **dh** presented in [13].

表 2 shows the algorithmic of **dh**. All data is distributed in p *workers* and the algorithm performas 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.

```
DH(IN ⊕, IN ⊗, INOUT data)
 1: if master then
 2:   par do
 3:     DH(⊕, ⊗, data);
 4:   end par
 5:   for n from 1 to log2(numChd) do
 6:     gather data to tmp;
 7:     for i from 1 to (len(numChd)) do
 8:       if ((i-1) % exp2(n))/2 = 0 then
 9:         Swap(tmp[i], tmp[i+exp2(n)])
10:       end if
11:     end for
12:     scatter tmp to list;
13:     par do
14:       if ((PID-1) % exp2(n))/2 = 0 then
15:         data := data ⊕ list;
16:       else
17:         date := data ⊗ list;
18:       end if
19:     end par
20:   end for
21: else
22:   for n from 1 to log2(len(data)) do
23:     for i from 1 to (len(data)) do
24:       if ((i-1) % exp2(n))/2 = 0 then
25:         tmp := data[i] ⊕ data[i+exp2(n)];
26:       else
27:         tmp := data[i-exp2(n)] ⊗ data[i];
28:       end if
29:     end for
30:     data := tmp;
31:   end for
32: end if
```

表 2 Distributable Homomorphism (dh)

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:

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

We can see from the pseudo code, **dh** imposes many "horizontal" communications. Our experience in teaching Master course "Parallel Algorithm" at Université Paris-Est Créteil [16] and the above example confirm our claims that **scatter** and **gather** can be used to program a large subset of all parallel functions. However, an optimal execution of all-to-all communication is still desirable. In the following section will discuss how to express such communication pattern on SGL.

4 The GPS Theorem

4.1 Gather-Scatter Communication

[14] shows that many basic algorithms can be efficiently and cleanly programmed with the model, while satisfying a simple performance model. But the case of the distributed homomorphism algorithm poses a fundamental problem for SGL. The tree topology architecture suggests a limited bandwidth around the master. Either this is experimentally true, in which case sample-sort cannot be generally efficient (for which SGL's simplicity could not be blamed) or the topology abstracts the actual communication bandwidth assuming that all links in the architecture have the same "width". This may hold if:

(a) indeed the network is a fat-tree; or

(b) the architecture's level below the master allows for horizontal communications that SGL does not express.

Case (a) is already covered by our SGL cost model but it was not observed in our experiments: the

master's bandwidth can be overwhelmed when an algorithm such as sorting moves a large subset of all data across the master. Case (b) is not only likely, but is the norm for flat architectures.

The pending issue with SGL is thus: can we program general communication patterns such as those involved in parallel sample-sort. The `if master` conditional selects an instruction branch depending on the local node's number of children: zero 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.

4.2 The GPS Theorem

Solutions to SGL's algorithmic incompleteness are proposed: a compromise between simple centralised communications and more general horizontal communications that are more difficult to program and error-prone. We start from the notion that we may program communications that are logically centralised but physically "horizontal". To support this claim we first present a theoretical result showing how successive supersteps can be partially compressed into a single horizontal communication-synchronization phase.

A first type of solution to SGL's lack of horizontal communications can come in the form of compiler optimisations. Sub-programs that concentrate data (gather), then locally process it on the master, then redistribute it (scatter) can be compiled to horizontal communications without the need for concentration. This possibility was not implemented so far for lack of a complete SGL language with syntax, compiler and code generation, a rather long subproject to develop. But we can prove that this kind of optimisation is possible in the form of a theorem based on our operational semantics.

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.

GPS-theorem

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 $\{1,\ldots,p\}$ 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 sub-programs must be evaluated 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, child 1, child 2, child of child 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'): $\sigma''' = \sigma''[\overrightarrow{V}/\sigma''(\widetilde{W})_i \ \ | i = 1,\ldots,p]$. We thus have:

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

□

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 sub-program $\mathcal{P}$ into π is beyond the scope of this chapter and would require a more complex language than what is covered by our current semantics for SGL.

4. 3 BSML Programming

BSML developed at *Université d'Orléans* and *Université Paris XII* (now called *Université Paris-Est Créteil*) is a library for OCaml implementing partially the Bulk Synchronous Parallel ML language [7]. There is in BSML an abstract polymorphic type α *par* which represents the type of p-wide parallel vectors of values of type α, one per process. It is very different from usual SPMD programming where messages and processes are explicit, and programs may be non-deterministic or may contain deadlocks. In fact a large subset of BSML parallel programs are purely functional. The newest version (0.5) of core BSML library is based on the following primitives:

mkpar : $(int \rightarrow \alpha) \rightarrow \alpha\ par$

proj : $\alpha\ par \rightarrow (int \rightarrow \alpha)$

apply : $(\alpha \rightarrow \beta)\ par \rightarrow \alpha\ par \rightarrow \beta\ par$

put : $(int \rightarrow \alpha)\ par \rightarrow (int \rightarrow \alpha)\ par$

The semantics of BSML primitives is described by the use of parallel values. Parallel value $<x_0, x_1, \ldots, x_{p-1}>$ represents a set of local values of a given type, such that x_i is stored on processor i and p is the number of processors.

In BSML, **mkpar** is the parallel constructor: **mkpar** f computes the value $<f\ 0,\ f\ 1, \ldots,\ f\ (p-1)>$. **proj** is the parallel destructor: **proj** $<x_0, x_1, \ldots, x_{p-1}>$ computes a function f such that (f i) = x_i. **apply** is the asynchronous parallel transformer: **apply** $<f_0,\ f_1, \ldots,\ f_{p-1}>$ $<x_0,\ x_1, \ldots, x_{p-1}>$ computes $<f_0x_0,\ f_1x_1, \ldots,\ f_{p-1}x_{p-1}>$. Finally, **put** is the synchronous (communicating) parallel transformer: **put** $<g_0, g_1, \ldots, g_{p-1}>$ computes a parallel vector of functions that contain the transported messages that were specified by the g_i. The input local functions are used to specify the outgoing messages thus: $g_i\ j$ is the value that processor i wishes to send to processor j. The result of applying **put** is a parallel vector of functions dual to the g_i: they specify which value was received from a given distant processor.

The execution of **mkpar** is purely local and so is the execution of the **apply** primitive. The execution of **proj** uses an all-to-all communication and the execution of **put** is a general BSP communication (any processor-processor relation can be implemented with it). Experience with BSML for more than a decade has shown that **proj** is much easier to use than **put** , that **proj** can be used to program a large subset of all parallel functions, but that algorithms such as sample-sort cannot be implemented without **put** .

4. 4 Simplifying BSML's PUT

We now propose one solution to SGL's lack of horizontal communication – the `GPS` function – for parallel programming to simplify BSML's general communication `put` function. It is based on the following simulation of SGL by BSML for flat BSP machines.

There is a natural correspondence between a two-level SGL machine and a BSML program:

- the *master* corresponds to all non-*par* types in the BSML program i.e. all values that are replicated on each processor;
- the *workers* correspond to all local elements of *par* types in the BSML program;
- **scatter** corresponds to **mkpar** (*seq-to-par*);
- **gather** corresponds to **proj** (*par-to-seq*); and
- **pardo** corresponds to **apply** .

An algorithm such as parallel sample-sort cannot be programmed with pure SGL primitives i.e. with only **mkpar**, **apply**, **proj** and without the general communication primitive **put**. Our proposed solution is a simplified form of **put** that leads to the same parallel performance but resembles a G;P;S program as in the GPS theorem presented before.

The general BSML communication primitive is **put** and it has the following type:

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

with the input parallel vector containing a *destination-value* map at each processor, and the output parallel vector containing a *sender-value* map at each processor.

The *simplified* version that we propose is called `sgl_gps` and has the following type:

```
(int -> 'a -> 'b list)
  -> (int -> 'b list -> 'c)
  -> 'a par -> 'c par
```

The first argument relates to data input for the communication phase. It specifies how each processor (rank an integer) splits a value into a list of p values, one for every destination. The second argument relates to data reception after communication. It specifies how each processor (rank an integer) aggregates a list of p received values into a local value. The third argument is the input parallel vector and the function produces its output parallel vector by applying a single communication-synchronisation superstep, like BSML's **put**.

The new function is considered to be simpler to be used than **put** because: (a) it separates meta-data (first two arguments) from the actual data to be communicated (third argument); and (b) the meta-data is sequential.

4.5 Experiments

表 3 shows the BSML implementation for the new communication function. Here **parfun** is an abbreviation for

```
fun f x -> apply (replicate f) x
```

provided by BSML's module *Bsmlbase*, where **replicate** is an abbreviation for

```
fun f -> mkpar (fun i -> f)
```

from the same module. This function is similar to List.map for parallel vectors:

parfun $f < x_0, \ldots, x_{(p-1)} > = < f\ x_0, \ldots, f\ x_{(p-1)} >$

And **bsp_p** is a machine parameter accessor of BSML giving the number p of processors in the parallel machine.

```
(* === BSML libraries === *)
open Bsml
open Base
open Tools

(* === Auxiliary functions === *)
let rec (from_to: int -> int -> int list
    ) = fun debut fin ->
  if debut > fin then [] else debut::(
      from_to (debut+1) fin);;

let (procs_list: int list) = from_to 0 (
    bsp_p-1) ;;

(* === SGL g-p-s function === *)
let (sgl_gps: (int -> 'a -> 'b list) ->
    (int -> 'b list -> 'c) -> 'a par ->
    'c par) = fun split assemble indata
    ->
   let splitted = apply (mkpar split)
       indata in
   let exchange = put (parfun List.nth
       splitted) in
   let permuted = parfun (fun l -> List.
       map l procs_list) exchange in
   apply (mkpar assemble) permuted
;;
```

表 3 Code of GPS function over BSML

We implemented Tiskin-McColl parallel sample-sort [22] with this new communication primitive in BSML, then experimented our **GPS** implementation on the French Tier-0 GENCI/TGCC Curie supercomputer's Thin nodes: B510 bullx. Each node is composed of two 8-core 2.7 GHz Inter Sandy Bridge EP (E5-2680) CPUs, and the nodes are interconnected via an InfiniBand QDR Full Fat Tree network. We fixed the total size of input data at 12 800 000 integer numbers, and varied the number of processors from 2 to 512. The execution time was measured by OCaml's function `Unix.gettimeofday`.

The formula of speedup we used is:

$$Speedup = \frac{ExeTime_{Seq}}{ExeTime_{Par}}$$

where $ExeTime_{Seq} = 47.024\ sec$ is based on OCaml's sequential function `List.sort` processing 12 800 000 random integer numbers generated by

the function `Random.int`.

Since the bound of sequential sorting algorithm is $O(n \log n)$, the formula for the efficiency we used is slightly different from the standard one which is simply $speedup \times \frac{1}{p}$. Here it is:

$$Efficiency = Speedup \times \frac{n \log n}{m \log m}$$

where n is the size of input data per processor and m is the total size of data: 12 800 000. This formula is coherent with Shi & Schaeffer's method presented in the paper [20] introducing Parallel Sorting by Regular Sampling (PSRS). Tiskin-McColl's algorithm is a BSP adaptation of PSRS.

表 4 shows the results of our experiment using 2 to 64 processors. The efficiency is about 0.80 until the communication cost overtakes the computation cost when we use a large parallel system to compute a small amount of number. That's why the efficiencies with 128, 256 and 512 processors are decreased to 0.34, 0.05 and 0.009.

# of proc	2	4	8	16	32	64
Time (s)	27.3	14.3	7.5	3.7	1.9	1.1
Speedup	1.7	3.3	6.2	12.8	25.2	44.4
Efficiency	0.86	0.80	0.78	0.80	0.79	0.69

表 4 Speedup and efficiency of parallel sample-sort in GPS implementation

We also tested on the same machine the same sorting algorithm implemented in standard BSML (with **put**), which can be found in L. Gesbert's PhD thesis [8]. We used the same method as before for measuring the execution time, speedup and efficiency of this **put** implementation. The results are shown in 表 5.

図 2 shows the speedup of the **GPS** implementation and the **put** implementation of parallel sample-sort. Our measurement shows that the **GPS** implementation is as fast as the **put** implementation. Furthermore, the **GPS** implementation

# of proc	2	4	8	16	32	64
Time (s)	27.5	14.3	7.5	3.7	1.9	1.1
Speedup	1.7	3.3	6.2	12.7	25.3	44.2
Efficiency	0.85	0.82	0.78	0.79	0.79	0.69

表 5 Speedup and efficiency of parallel sample-sort using BSML put

keeps the same speedup as the standard BSML one, because the sorting algorithms are the same for both implementations. These results confirm our claim that **GPS** can simplify programming while keeping a good performance.

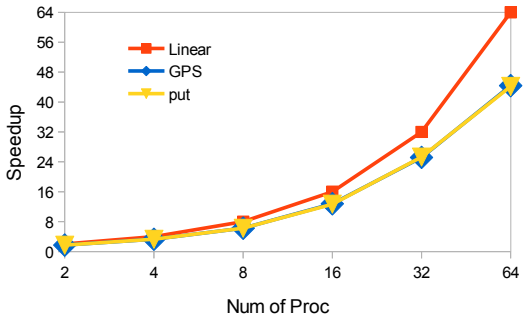

図 2 Speedup of GPS and put parallel sample-sort implementations

The main difference between these two implementations is that the **sgl_gps** function provides a sequential view for coding. Developers need only to focus on how to split the local data and how to assemble the received data on a local node. The all-to-all communication and synchronisation are performed implicitly by **sgl_gps**. The cost of communication may be optimised during compilation (if the compiler supports natively the **sgl_gps** function). Contrariwise, the **put** function focusses on the communication. A *destination-value* map for each processor before **put** and a *sender-value* map for each processor after **put** must be designed by

the developer using **mkpar**, **apply**, etc. Moreover, developers must work on a parallel environment even before applying **put** itself.

5 Conclusion

The definition of SGL is only a first step towards its safe and efficient application to high-level parallel programming. Adding a general communications primitive to SGL may be a radical solution. But this would have hurt the practical and theoretical simplicity of SGL. Experience and research with BSML shows that the semantics and practical use of such a primitive (**put**) is more complex and error-prone than pure SGL. This is obvious because the semantics of **put** is based on a communication matrix, while **scatter** and **gather** are described by 1-dimensional vectors of messages.

Further analysis has led us to propose two elements of a general solution to this dilemma. The first one is the GPS theorem: a semantic equivalence between a **gather**; `P`; **scatter** sub-programs (where `P` is a local sequential program in the *master*) and horizontal **put**-like operations. This result is the basis of future compiler optimisations whereby a family of clean but inefficient SGL programs can be compiled to lower-level but more efficient programs using horizontal communications. The second proposed solution is a simplified form of **put** that we have designed and experimented in BSML. With it, the Tiskin-McColl BSP samplesort algorithm is programmed without having to encode a general communication matrix but only its sender-side 1-to-n and receiver-side n-to-1 communication relations.

As a result our overall position w.r.t. the communication primitives is to promote SGL-style simplified programming for most programs, while allowing to "escape" in the more complex use of horizontal communication with the GPS function. The GPS theorem reduces the need for the latter and the function's design makes it easier to use than BSML's **put** function.

Future work will experiment multi-level and heterogeneous versions of this extension to SGL. A study on automatic detection of machine's communication throughput with a formal network performance formula is also desired.

謝辞

The first author thanks EXQIM company for funding his research for the SGL model during 2009 to 2013 with French National Association of Research and Technology. The second author thanks W.F. McColl for introducing him to the BSP model in the early 1990s. SGL is inspired by the work of colleagues who are too numerous to cite here, but special thanks are due to Prof. Frédéric Loulergue et al. for their work on BSML. Both authors thank Partnership for Advanced Computing in Europe (PRACE) Research Infrastructure for accessing the Curie supercomputer for our experiments through the MHBSP project (RA1634). We thank sincerely Prof. Zhenjiang Hu for supporting the presentation of this paper on JSSST 2014.

参 考 文 献

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

[2] Barker, K. J., Davis, K., Hoisie, A., Kerbyson, D. J., Lang, M., Pakin, S., and Sancho, J. C.: Entering the petaflop era: the architecture and performance of Roadrunner, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, IEEE Press, 2008, pp. 1:1–1:11.

[3] Bougé, L.: The data parallel programming model: A semantic perspective, *The Data Parallel Programming Model*, Perrin, G.-R. and Darte, A.(eds.), Lecture Notes in Computer Science, Vol. 1132, Springer Berlin Heidelberg, 1996, pp. 4–26.

[4] Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming, *Parallel Computing*, Vol. 30, No. 3(2004), pp. 389–406.

[5] Falcou, J., Serot, J., Chateau, T., and Lapreste, J. T.: QUAFF : Efficient C++ Design for Parallel Skeletons, *Parallel Computing*, Vol. 32, No. 7-8(2006), pp. 604–615.

[6] Fortune, S. and Wyllie, J.: Parallelism in random access machines, *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, New York, NY, USA, ACM, 1978, pp. 114–118.

[7] Gava, F.: *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs. Sémantiques, implantations et certification*, PhD Thesis, Université Paris XII-Val de Marne, LACL, 2005.

[8] Gesbert, L.: *Développement systématique et sûreté d'exécution en programmation parallèle structurée*, PhD Thesis, Université Paris-Est, 2009.

[9] Johns, C. R. and Brokenshire, D. A.: Introduction to the cell broadband engine architecture, *IBM J. Res. Dev.*, Vol. 51(2007), pp. 503–519.

[10] Kuchen, H. and Cole, M.: The Integration of Task and Data Parallel Skeletons, *Parallel Processing Letters*, Vol. 12, No. 2(2002), pp. 141–155.

[11] Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P.: Heterogeneous Chip Multiprocessors, *Computer*, Vol. 38(2005), pp. 32–38.

[12] Li, C.: *A Software-Hardware Bridging Model for Simplifying Parallel Programming*, PhD Thesis, Université Paris-Est, 2013.

[13] Li, C., Gava, F., and Hains, G.: Implementation of Data-Parallel Skeletons: A Case Study Using a Coarse-Grained Hierarchical Model, *Proceedings of the 2012 11th International Symposium on Parallel and Distributed Computing*, ISPDC '12, Washington, DC, USA, IEEE Computer Society, 2012, pp. 26–33.

[14] Li, C. and Hains, G.: A simple bridging model for high-performance computing, *High Performance Computing and Simulation, 2011 International Conference on*, HPCS 2011, Washington, DC, USA, IEEE Computer Society, 2011, pp. 249–256.

[15] Li, C. and Hains, G.: SGL: towards a bridging model for heterogeneous hierarchical platforms, *Int. J. High Perform. Comput. Netw.*, Vol. 7, No. 2(2012), pp. 139–151.

[16] Li, C. and Hains, G.: GPS: Towards Simplified Communication on SGL Model, *Proceedings of IEEE International Symposium on Parallel & Distributed Processing, Workshops*, IPDPS '14, Washington, DC, USA, IEEE Computer Society, 2014.

[17] Loulergue, F.: *Conception de langages fonctionnels pour la programmation massivement parallèle*, thèse de doctorat, Université d'Orléans, LIFO, 4 rue Léonard de Vinci, BP 6759, F-45067 Orléans Cedex 2, France, January 2000.

[18] Loulergue, F., Hains, G., and Foisy, C.: A Calculus of Functional BSP Programs, *Science of Computer Programming*, Vol. 37, No. 1-3(2000), pp. 253–277.

[19] McColl, W. F. and Walker, D.: Theory and Algorithms for Parallel Computation, *Euro-Par*, Pritchard, D. J. and Reeve, J.(eds.), Lecture Notes in Computer Science, Vol. 1470, Springer, 1998, pp. 863–864.

[20] Shi, H. and Schaeffer, J.: Parallel sorting by regular sampling, *J. Parallel Distrib. Comput.*, Vol. 14, No. 4(1992), pp. 361–372.

[21] staff, I. j. o. R. and Development: Overview of the IBM Blue Gene/P project, *IBM J. Res. Dev.*, Vol. 52(2008), pp. 199–220.

[22] Tiskin, A.: *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*, PhD Thesis, University of Oxford, 1999.

[23] Valiant, L. G.: A bridging model for parallel computation, *Commun. ACM*, Vol. 33(1990), pp. 103–111.

[24] Winskel, G.: *The formal semantics of programming languages: an introduction*, MIT Press, Cambridge, MA, USA, 1993.