# A simple Bridging Model for High-Performance Computing

Chong Li, Gaétan Hains


# A simple Bridging Model for High-Performance Computing

**Chong LI** **Gaétan HAINS**

*December 2010*

**TR–LACL–2010–12**

**Laboratoire d'Algorithmique, Complexité et Logique (LACL)**
**Université de Paris-Est Créteil (UPEC), Faculté des Science et Technologie**
61, Avenue du Général de Gaulle, 94010 Créteil cedex, France
Tel.: (+33)(1) 45 17 16 47, Fax: (+33)(1) 45 17 66 01

Laboratory for Algorithmics, Complexity and Logic (LACL)
University of Paris-Est Créteil (UPEC)

Technical Report **TR–LACL–2010–12**

C. LI, G. HAINS

*A simple bridging model for high-performance computing*

# A simple bridging model for high-performance computing

Chong Li[1,2] Gaétan Hains[1,2]

[1] LACL, Université Paris-Est
61, Avenue du Général de Gaulle, 94010 Créteil, France

[2] EXQIM S.A.S.
24, Rue de Caumartin, 75009 Paris, France

chong.li@exqim.com gaetan.hains@u-pec.fr

**Abstract**

This report introduces the Scatter-Gather parallel-programming and parallel execution model in the form of a simple imperative language named SGL. Its design is based on past experience with Bulk-synchronous parallel (BSP) programming and BSP language design. SGL's novel features are motivated by the last decade's move towards multi-level and heterogeneous parallel architectures invoving multi-core processors, graphics accelerators and hierarchical routing networks in the largest multiprocessing systems. The design of SGL is coherent with Valiant's Multi-BSP while offering a programming interface that is even simpler than the primitives of Bulk-Synchronous parallel ML (BSML). SGL appears to cover a large subset of all BSP algorithms while avoiding complex message-passing programming. It allows automatic load balancing and like all BSP-inspired systems, predictable, portable and scalable performance.

# Contents

# 1 Introduction

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.

From a programming point of view, paper [Adv09] gives a perspective on work collectively spanning approximately 30 years. It shows how difficult it is to formalize the seemingly simple and fundamental property of "what value a read should return in a multithreaded program". Safe languages must be given semantics that computer science graduates and developers can understand with a reasonable effort. The author of this survey believes we need to rethink higher-level disciplines that make it much easier to write parallel programs and that can be enforced by our languages and systems. As we move toward more disciplined programming models, there is also a new opportunity for a hardware/software co-designed approach that rethinks the hardware/software interface and the hardware implementations of all concurrency mechanisms.

As the above remark highlights, multitreaded *semantics* is far too complex for realistic software development. Yet parallel execution is synonymous with multiple processes or multithreading, without which there can be no parallel speed-up. So how should programmers avoid the complexity of multi-threaded programming and yet expect scalable performance? Part of the answer comes from the observation that the vast majority of parallel algorithms are deterministic. Along this line of reasoning, researchers like M. Cole [Col89] and H. Kuchen have developed the paradigm of *algorithmic skeletons* for deterministic and deadlock-free parallel programming. Skeletons are akin to design patterns for parallel execution. A large body of programming research literature supports the view that most if not all parallel application software should be based on families of algorithmic skeletons.

A deterministic and high-level parallel programming interface is indeed a major improvement over explicit message passing. But the diminished expressive power of skeletons is not only an advantage. Unlike sequential equivalents, skeletons are not libraries in the classical sense because their host language (e.g. C) is necessarily less expressive than the language in which they are written (e.g. C+MPI). This is due to the lack of a base language that is not just Turing-complete but complete for parallel algorithms, a notion that has not even been well defined yet. As a result there is no fixed notion of a set of skeleton *primitives* but instead the message-passing primitives used to implement them. That feature of skeleton languages is similar to that of a language with automatic memory management: if the only concern is time complexity then a simpler automatic-memory language is sufficient to implement it; but if space complexity is to be explicit, then it is necessary to use an explicit-memory allocation language to implement the original one.

These remarks gave rise to our notion of *explicit processes*: without an explicit notion of the number of physical parallel processes, parallel speed-up is not part of programming semantics. A language that is expected to express parallel algorithms must also express not a function from computation events to physical units (this function may not be injective ...) but just the inverse. In [HF93] the second author introduced this notion of explicit processes through a deterministic parallel dialect of ML called DPML: the program's semantics is parametrized on the number of physical processes and their local indexes (processor ID's as they are often called). We concluded that DPML can serve as implementation language for skeletons, yet it remains a deterministic language.

Meanwhile, a major conceptual step was taken by L. Valiant [Val90] who introduced his Bulk-Synchronous Parallel (BSP) model. Inspired by complexity theory's PRAM model of parallel computers, Valiant proposed that parallel algorithms be designed and measured by accounting not only for the classical balance between time and parallel space (the number of processors) but also for communication and synchronization. A BSP computation is a sequence

of so-called supersteps. Each superstep combines asynchronous local computation with point-to-point communications that are coordinated by a global synchronization to ensure coherence and determinism. The resulting performance model is both realistic and tractable so that researchers like McColl et al. [MW98] 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. A well-known implementation of BSP is the Paderborn University BSP library (or PUB lib) [BJvOR03]. This work has been extended by many similar models: LogP [CKP+93], LogGP [AISS95] for long messages, HLogGP [BP04] for heterogeneous processors, LogGPO [CZZZ09] to account for computation-communication overlap and the CGM model [DFRC93].

From the mi-1990's it became clear that BSP was very well adapted for implementing algorithmic skeletons: its view of the parallel system included explicit processes and added a small set of network performance parameters to allow for predictable performance. Our earlier language design DPML was still too flexible to be restrained to the class of BSP executions. Loulergue, Hains and Foisy then designed BS-lambda [LHF00] as a minimal model of computation with BSP operations (an extension of the lambda-calculus). BS-lambda became the basis for Bulk-Synchronous ML (BSML) [Lou00] [Lou10] a variant of CAML under development by Loulergue et al. since 2000. BSML improves on DPML by offering a purely functional semantics, and a much 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 from program semantics and the BSP parameters of the host architecture.

Much langage- and software-engineering research has been conducted around BSML: weak desynchronization[LGAD04], a grid-computing extension, program proofs, deterministic exception handling, a C++ variant of the language, but this is beyond the scope of this report. With respect to BSML, our new SGL *Scatter-Gather Language* model can be understood as 1. a generalization to hierarchical architectures by breaking the restriction to a flat vector of processors 2. a simplified set of primitives where the "all-to-all" processor relations of `put` are replaced by simpler "one-to-all" and "all-to-one" relations. Our presentation of SGL is in the form of an imperative langague while BSML is a variant of ML hence a functional language. But that last difference is less important than the above two.

Among the earlier works on BSML, the grid- or *departmental meta-computing ML* or DMML model and language of Gava and Loulergue [GL05] described a two-level BSP. Its lower level represents local parallel clusters and its top level a grid of such systems. SGL is both a generalization of this design, for systems that have potentially more levels and a restriction of the programming interface, by avoiding the most general communication operator `put`. SGL can cover at lower levels inside the machines and also higher levels or more up, for example, from multi-core processors like Nvidia Tesla or STI Cell, to giant systems like Tera-100 or Blue Gene etc. An analysis of the exact differences between both models remains to be done but the experience gained on DMML will be reused for investigating and developing SGL.

The next section gives precise arguments that motivate our SGL model, in the context of existing similar models like BSP or LogP. Then we describe the SGL model's abstract machine, execution- and cost-model to relate machine parameters with actual algorithm performance. The section 4 defines formally the programming model as the operational semantics for an imperative mini-language. Then we present and analyze some performance-measurement experiments for primitive algorithms. Finally we conclude with a summary of ongoing work and open problems related to SGL.

## 2 Motivations

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 Blue Gene/L [ABB+03], Blue Gene/P [joRs08] feature multi-processors on one card, multi-core processors on one chip, multiple-rack clusters etc. The Cell/B.E. [KDH+05, JB07], Cell-based RoadRunner [BDH+08] and GPU's feature a CPU with Master-Worker architecture. Moreover, [KTJR05] observes that heterogeneous chip multiprocessors present unique opportunities for improving system throughput and reducing processor power. 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.

Towards the middle of the 2000's decade it was obvious that models like BSP should adapt or generalize to the new variety of architectures. Yet, programming simplicity and performance portability should be retained as much as possible. With these goals in mind, Valiant introduced Multi-BSP [Val08] a multi-level variant of the BSP model 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. In that sense it preserves the notion of explicit processes and, as we will show in this report with our SGL design, allows to solve three pending problems with BSP programming:

1. The flat nature of BSP is not easily reconciled with divide-and-conquer parallelism [Hai98], yet many parallel algorithms (e.g. Strassen matrix multiplication, quad-tree methods etc.) are highly artificial to program any other way than recursively.

2. BSP was designed natively to scale out with the number of processors, and it was assumed the clock speeds of CPU will continue to improve. However, there are barriers to further significant improvements in operating frequency due to voltage leakage across internal chip components and heat dissipation limits[SMD+10]. The parallelism is thus used to scale up the computing power nowadays using multithreaded cores, multicore CPUs, Cell processors, GP-GPU, etc. We do not know how to design algorithms for nested level systems with the original flat BSP model. The hierarchical architectures share communication resources inside every level but not between different levels. The BSP cost model is not suitable for this kind of architectures.

3. After teaching BSML programming for more than 10 years at the Universities of Orleans and Paris-Est, we observe that many fourth-year[1] students can master the constructor (mkpar), destructor (proj), and asynchronous parallel transformation (apply) while having much trouble with the general communication primitive (put).

Based on all the above developments and the last three observations we will now define the SGL model which realizes a programming model for Multi-BSP, generalizes it to heterogeneous systems and yet simplifies the BSML primitives to a set of three. SGL 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). We show how to embed this model in an imperative language, how to define its operational semantics, its performance model, how to use it to code useful algorithms (reduction, parallel scan and merge sort). We then provide some initial measurements to validate its performance model and explanations as to how it can easily adapt to heterogeneous systems.

The work summarized here is initial evidence that SGL can provide both a simpler programming interface than BSML and a richer execution model, like Multi-BSP, to allow for automatic tuning on most types of modern architectures.

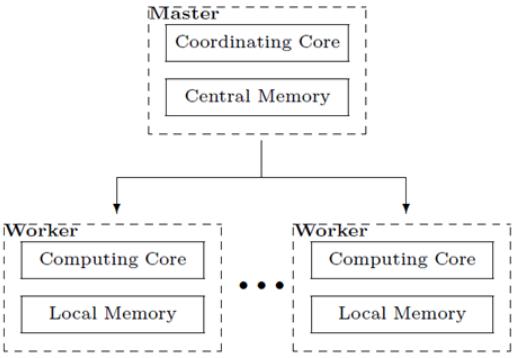
[1] French "Bac+4" or M1-level which is equivalent to final-year BSc

# 3 The SGL Model

The PRAM model [GR88] 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 controler (as in a SIMD architecture). The BSP model [Val90] 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 synchronization 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. But 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 research popular in the 1980: algorithms for specific interconnect topologies like hypercube algorithms, systolic algorithms, rectangular grid algorithms etc. We mostly 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 we propose does introduce a degree of topology on the set of processors. But this topology is purely hierarchical and not intended to become an explicit factor of algorithm diversity. Just like PRAM/BSP algorithms use the $p$ parameter to adapt to systems of all sizes, our SGL algorithms will use the machine parameters to adapt to all possible systems.

## 3.1 The SGL computer or abstract machine

The SGL model begins with a set of sequential processors composed of a computation element ("core") and memory unit.

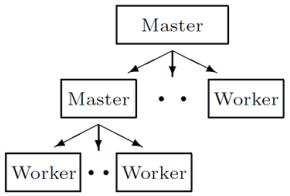



The processors are arranged in a tree structure with the root being called a "master" and its sons that are either masters themselves or leaf- "workers". The number of worker-sons is not limited so that the BSP/PRAM concept of a flat $p$-vector of processors is easily recovered in SGL. Different forms are possible:

**(1)** Only 1 worker without master = A sequential machine

**(2)** Master + workers = A parallel machine, e.g. a BSP computer

**(3)** A hierarchical machine



The logical structure of a SGL computer satisfies the following constraints.

- A system shall have one and only one *root-master*.
- A *master* coordinates its *children*.
- A *worker* shall be controlled by one and only one *master*[2].
- Communication is always between a *master* and its *children*.

### 3.2 Execution Model

An SGL *program* is composed of a sequence of *super-steps*. Each *super-step* is composed of 4 phases [3] :

- a scatter communication phase initiated by the master
- an asynchronous computation phase performed by the children. Note: A child computation phase can also be a *super-step*.
- a gather communication phase centered on the master
- a local computation phase on the master

One can usually write an SGL program recursively:

```
if the node is master and its children are to be used, then
      split the input data into blocks
      scatter data blocks
      compute on individual data blocks in children (parallel)
      gather children's results
      compute children results in master (if necessary)
else
      compute on local data directly in master/worker
fi
```

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.

### 3.3 Cost Model

Like all such 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 algorithms (i.e. of its execution on given input data) is the sum of the costs of its supersteps. That follows from the understanding that supersteps execute sequentially. The cost of an individual superstep is split into two independent terms: computation cost and communication cost.

[2] *Masters* can be replicated by underlying libraries for fault-tolerance.

[3] The initial computing data and final result after computing can be either distributed in workers or centralized in root-master.

$$Cost_{total} = \sum Cost_{supstep}$$

$$Cost_{supstep} = Comp_{total} + Comm_{total}$$

Computation cost is the sum of local computation times 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 work in children. Communication cost is split into the times for performing the scatter and the gather operation.

$$= Comm_{scat} + \max Comp_{children} + Comm_{gath} + Comp_{master}$$

$$\begin{aligned} Comm_{scat} &= Words_{scat} * Bwidth_{scat} + Synchronization \\ Comm_{gath} &= Words_{gath} * Bwidth_{gath} + Synchronization \\ Comp &= Instructions/ClkRate \end{aligned}$$

Finally, communication costs are estimated by a linear term similar to BSP's $g.h + L$, 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 asymtotic complexity classes, while portable concrete SGL algorithms can be analyzed 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.

We remind the reader that, like the machine architecture, all cost formulae above are intended to be recursive. The local computation cost on a child node can itself be an SGL algorithm cost if the machine structure is such.

### 3.4 Parameters and detailed cost formulae

Concrete cost estimates and measurements use the following machine- and algorithm parameters.

- **l** : the latency to perform a *scatter* or *gather* communication synchronization i.e. the time to execute a 1-bit scatter or 1-bit gather.
- **g** : the gap, $g_\downarrow$ defined as the minimum time interval for transmiting one word from *master* to its *children*, and $g_\uparrow$ for *children* to its *master*. We use a single $g$ for symmetric communication.
- **k** : number of words to transmit, $k_\downarrow$ denotes number of words that *master* scatter to *children*, and $k_\uparrow$ denotes number of words that *master* gather from *children*.
- **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 its *children*. This parameter without index, $w$ refers to a local quantity.
- **c** : computation speed of processors, $c_0$ denotes the time interval for performing a unit of *work* in *master* and $c_i(i{=}1..p)$ denotes the time interval for performing an unit of *work* in *children* processor. This parameter without index, $c$ refers to a local quantity.
- **p** : the number of children processors that *master* has.

In general the cost formulae are:

$$\begin{aligned} Cost_{Master} &= \max_{i=1..p}(Cost_{child_i}) + w_0 * c_0 + k_\downarrow * g_\downarrow + k_\uparrow * g_\uparrow + 2l \\ Cost_{Worker} &= w_i * c_i \end{aligned}$$

which clearly covers the possibility of a heterogeneous architecture.
More often, but not necessarily, we have symmetric communication:

$$Cost_{supstep} = w * c + [\max_{i=1..p}(Cost_{chd_i}) + (k_{\downarrow} + k_{\uparrow}) * g + 2l]$$

# 4 Programming Model

SGL's precise definition is an imperative language for which we now give an operational semantics.

## 4.1 Operational Semantics

We enrich Winskel's basic imperative language IMP [Win93] to yield our deterministic parallel programming language - SGL:

### 4.1.1 Syntax

**Values** 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.

- $n$ ranges over numbers **Nat**
- $< n_1, n_2, \ldots n_\ell >$ ranges over arrays of number **Vec**
  Here $\ell$ denotes length of array and $\ell \in Nat$
- $< v_1, v_2, \ldots v_\ell >$ ranges over arrays of array **VecVec**

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.

**Locations** 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

**Operations**

- $\odot$ ranges over binary operations **Op** ::= $+ \mid - \mid * \mid /$

**Expressions** Expressions are relatively standard with the convenience of scalar-to-vector (sequential) operations.

- $a$ ranges over scalar arithmetic expressions **Aexp** ::= $n \mid X \mid a \odot a \mid \overrightarrow{V}[a]$
- $b$ ranges over scalar boolean expressions **Bexp** ::= **true** | **false** | $a = a \mid a \leq a \mid \neg b \mid b \wedge b \mid b \vee b$
- $v$ ranges over vectorial expressions **Vexp** ::= $< a_1, a_2, \ldots a_\ell > \mid \overrightarrow{V} \mid v \odot a \mid v \odot v \mid \widetilde{W}[a]$
- $w$ ranges over vectorial vectorial expressions **VVexp** ::= $< v_1, v_2, ..v_\ell > \mid \widetilde{W}$

**Commands** The language's commands include classical sequential constructs with SGL's 3 primitives scatter, pardo and gather. Their exact meanings 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** $c$ **else** $c$
- Auxiliary commands **Aux** ::= **numChd** | **len** $\overrightarrow{V}$ | **len** $\widetilde{W}$

### 4.1.2 Environments

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.

**Functions in States $\Sigma$**

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

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.

- $\sigma : NatLoc \to Pos \to Nat$, thus $\sigma(X_{pos}) = \sigma_{pos}(X) \in Nat$
- $\sigma : VecLoc \to Pos \to Vec$, thus $\sigma(\overrightarrow{V}_{pos}) = \sigma_{pos}(\overrightarrow{V}) \in Vec$

**Rules of Vectorial Expressions** The semantics of vector expressions is standard and deserves no special explanations.

$$\frac{\forall_{i=1..\ell}\langle a_i,\ \sigma\rangle \to n_i}{\langle < a_1, a_2, \ldots a_\ell >,\ \sigma\rangle \to < n_1, n_2, \ldots n_\ell >}$$

$$\langle \overrightarrow{V},\ \sigma\rangle \to \sigma(\overrightarrow{V}),\ \textit{a value of the form} < n_1, n_2, \ldots n_\ell >$$

$$\frac{\langle \widetilde{W},\ \sigma\rangle \to \sigma(\widetilde{W}),\ \textit{a value of the form} < v_1, v_2, \ldots v_\ell > \quad \langle a,\ \sigma\rangle \to n}{\langle \widetilde{W}[a],\ \sigma\rangle \to v_n}$$

$$\frac{\langle v,\ \sigma\rangle \to < n'_1, n'_2, \ldots n'_\ell > \quad \langle a,\ \sigma\rangle \to n \quad \forall_{i=1\ldots\ell}\langle n'_i \odot n,\ \sigma\rangle \to n_i}{\langle v \odot a,\ \sigma\rangle \to < n_1, n_2, \ldots n_\ell >}$$

$$\frac{\langle v_1,\ \sigma\rangle \to < n'_1, n'_2, \ldots n'_\ell > \quad \langle v_2,\ \sigma\rangle \to < n''_1, n''_2, \ldots n''_\ell > \quad \forall_{i=1\ldots\ell}\langle n'_i \odot n''_i,\ \sigma\rangle \to n_i}{\langle v_1 \odot v_2,\ \sigma\rangle \to < n_1, n_2 \ldots n_\ell >}$$

*(Note: length of $v_1$ and length of $v_2$ shall be equal.)*

Similarly for the rules of the other expressions.

**Auxiliary Commands** The following predefined functions are necessary for using the parallel primitives. The first one is relative to the SGL machine structure.

- $\langle \textbf{numChd},\ \sigma\rangle \to n$
  return number of children that the processor has
- $\langle \textbf{len}\ \overrightarrow{V},\ \sigma\rangle \to n$
  return the length of $\overrightarrow{V}$
- $\langle \textbf{len}\ \widetilde{W},\ \sigma\rangle \to n$
  return the length of $\widetilde{W}$

**Primitive Commands**

- Skip
$$\langle \textbf{skip},\ \sigma\rangle \to \sigma$$
- Assignments
$$\frac{\langle a,\ \sigma\rangle \to n}{\langle X := a,\ \sigma\rangle \to \sigma[n/X]}$$
$$\frac{\langle v,\ \sigma\rangle \to < n_1, n_2, \ldots n_\ell >}{\langle \overrightarrow{V} := v,\ \sigma\rangle \to \sigma[< n_1, n_2, \ldots n_\ell > / \overrightarrow{V}]}$$
$$\frac{\langle w,\ \sigma\rangle \to < v_1, v_2, \ldots v_\ell >}{\langle \widetilde{W} := w,\ \sigma\rangle \to \sigma[< v_1, v_2, \ldots v_\ell > / \widetilde{W}]}$$
- Sequencing
$$\frac{\langle c_1,\ \sigma\rangle \to \sigma'' \quad \langle c_2,\ \sigma''\rangle \to \sigma'}{\langle c_1; c_2,\ \sigma\rangle \to \sigma'}$$
- Conditionals
$$\frac{\langle b,\ \sigma\rangle \to \textbf{true} \quad \langle c_1,\ \sigma\rangle \to \sigma'}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\ \sigma\rangle \to \sigma'}$$
$$\frac{\langle b,\ \sigma\rangle \to \textbf{false} \quad \langle c_2,\ \sigma\rangle \to \sigma'}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\ \sigma\rangle \to \sigma'}$$
- For-Loops
$$\frac{\langle X := a_1,\ \sigma\rangle \to \sigma'' \quad \langle \textbf{if}\ X \le a_2\ \textbf{then}\ \{c;\ X := X + 1;\ \textbf{for}\ X\ \textbf{from}\ X\ \textbf{to}\ a_2\ \textbf{do}\ c\}\ \textbf{else skip},\ \sigma''\rangle \to \sigma'}{\langle \textbf{for}\ X\ \textbf{from}\ a_1\ \textbf{to}\ a_2\ \textbf{do}\ c,\ \sigma\rangle \to \sigma'}$$
- Scatters
$$\frac{\langle w,\ \sigma\rangle \to < v_1, v_2, \ldots v_\ell > \quad \forall_{i=1..numChd} \langle \overrightarrow{V}_i := v_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter}\ w\ \textbf{to}\ \overrightarrow{V},\ \sigma\rangle \to \sigma'}$$
$$\frac{\langle v,\ \sigma\rangle \to < n_1, n_2, \ldots n_\ell > \quad \forall_{i=1..numChd} \langle X_i := n_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter}\ v\ \textbf{to}\ X,\ \sigma\rangle \to \sigma'}$$
- Gathers
$$\frac{\langle \widetilde{W} :=< \overrightarrow{V}_1, \overrightarrow{V}_2, \ldots \overrightarrow{V}_{numChd} >,\ \sigma\rangle \to \sigma'}{\langle \textbf{gather}\ \overrightarrow{V}\ \textbf{to}\ \widetilde{W},\ \sigma\rangle \to \sigma'}$$
$$\frac{\langle \overrightarrow{V} :=< X_1, X_2, \ldots X_{numChd} >,\ \sigma\rangle \to \sigma'}{\langle \textbf{gather}\ X\ \textbf{to}\ \overrightarrow{V},\ \sigma\rangle \to \sigma'}$$

- Parallel

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

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

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

# 5 Experimentation

Now that the language's cost model and operational semantics have been defined, we summarize some performance measurements and analyze them with respect to the cost model.

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.

Machine - SGI Altix ICE 8200EX :
Node - 16 computing nodes, each node has 2 CPUs
CPU - Intel Xeon E5440 (Quad-core, 2.83 GHz, FSB: 1333 MT/s, L2 Cache: 12 MB)
RAM - 512 GB in total (4 GB per core, DDR2-667 1.5ns)
Network - two 4X DDR InfiniBand switches (16 Gbit/s)

We build a 2-level SGL abstract machine to represent this physical machine:

| Unit | Children | Communication |
|---|---|---|
| Root-master | 16 nodes | InfiniBand |
| Node-master | 8 cores | Front-Side Bus |
| Worker | 0 | N/A |

## 5.1 Parameters Measurement

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 2.01). In the following table, the 4 first lines were used during our experiments of SGL, and the 4 last lines are only for comparing with BSP.

| Machine | Num of Proc | L ($\mu s$) | $g_\downarrow$ ($\mu s/32bits$) | $g_\uparrow$ ($\mu s/32bits$) |
|---|---|---|---|---|
| 2 nodes x 1 core | 2 | 1.48 | 0.00138 | 0.00215 |
| 4 nodes x 1 core | 4 | 2.85 | 0.00169 | 0.00200 |
| 8 nodes x 1 core | 8 | 4.37 | 0.00189 | 0.00205 |
| 16 nodes x 1 core | 16 | 5.96 | 0.00204 | 0.00209 |
| 16 nodes x 2 cores | 32 | 7.62 | 0.00214 | 0.00209 |
| 16 nodes x 4 cores | 64 | 7.93 | 0.00263 | 0.00211 |
| 16 nodes x 6 cores | 96 | 8.81 | 0.00288 | 0.00213 |
| 16 nodes x 8 cores | 128 | 9.89 | 0.00301 | 0.00277 |

Note: *In above table we see with SGI's MPI implementation, the cost of g increases with the number of processors, and for MPI_Gatherv there is a threshold around 2 ns/32bits.*

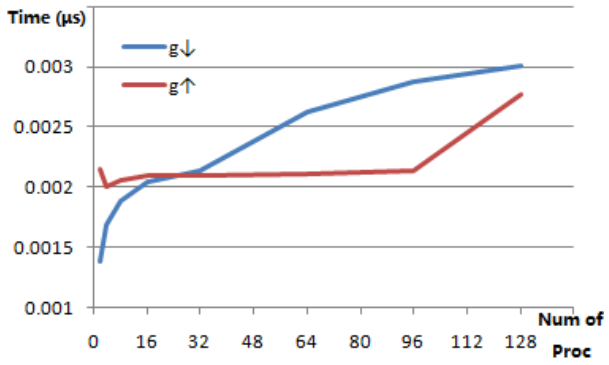



Figure 1: Measurement of $g$ in MPI

At the core (worker) level, we use OpenMP's Barrier for $L$ and the C language's function *memcpy* [4] for $g$:

| Machine | L ($\mu s$) | g ($\mu s/32bits$) |
|---|---|---|
| 2 cores | 12.08 | 0.00059 |
| 4 cores | 25.64 | 0.00059 |
| 6 cores | 37.80 | 0.00059 |
| 8 cores | 52.00 | 0.00059 |

According to the above tables, if we used flat BSP instead of the SGL model to represent our machine, the communication cost between *root-master* and *workers* would increase by nearly 0.4 $\mu s/32bits$: To represent our physical machine in BSP, we would build a 128-processor abstract machine using MPI. Under this architecture, the value of $g$ would be $\max(0.00301, 0.00277) = 0.00301\mu s/32bits$. In SGL, we built instead a 16-processor network at node level using MPI and a 8-processor network at core level using OpenMP. Thus, the value of $g_{\downarrow}$ is $0.00204 + 0.00059 = 0.00263\mu s/32bits$ and the value of $g_{\uparrow}$ is $0.00209 + 0.00059 = 0.00268\mu s/32bits$. This is a medium-scale but concrete example of practical gains obtained from the hierarchical model's better fit of physical architectures.

## 5.2 Algorithms

We have tested SGL variants of some of the most important basic parallel algorithms: parallel reduction (with the product operation), parallel prefix reductions also called scan (with the sum operation) and a sorting algorithm. For each one we wrote an SGL algorithm, implemented it by hand with the MPI/OpenMP operations mentionned above, and then compared the model's predicted vs observed run time for increasing data sizes. Finally we computed speed-up and parallel efficiency values.

The three tests illustrate SGL's relative programming simplicity on a relevant set of algorithms, convincing proof of the cost model's quality and initial proof that SGL programming has no intrinsic cost overhead.

### 5.2.1 Parallel Reduction

In this algorithm, each *worker* computes the product of its local scalar numbers. After that, the master fetches the computed product from its *children* and calculates the final product. Comments in braces are cost estimates for the corresponding lines.

[4] Here instead of transferring directly the pointers of data, we use memcpy() for replacing data in different memory regions to avoid concurrent access between CPU cores.

**Algorithm 1** Parallel Reduction

Reduction(IN $\overrightarrow{src}$, OUT $res$)
**begin**
1: **if master then**
2: **par do**
3: Reduction($\overrightarrow{src}$, $res$); $\{\Longleftarrow Reduction_{child_i}\}$
4: **end par**
5: **gather** $res$ **to** $\overrightarrow{lst}$; $\{\Longleftarrow p * g_{\uparrow} + l\}$
6: Product($\overrightarrow{lst}, res$) ; $\{\Longleftarrow O(p)\}$
7: **else**
8: Product($\overrightarrow{src}, res$); $\{\Longleftarrow O(n)\}$
9: **end if**
**end**

Product(IN $\overrightarrow{src}$, OUT $res$)
**begin**
$res := 1$;
**for** $i$ **from** 1 **to** (**len** $src$) **do**
$res := res * src[i]$;
**end for**
**end**

Line 3 is a recursive call to the algorithm and line 8, the no-children case, is a local sequential loop. The cost of the super-step (subscripts $\langle\rangle$ refer to the pseudo-code line):

$$Cost_{Master} = \max_{i=1..p}(Reduction_{Child_i\ \langle 3\rangle}) + O(p)\ _{\langle 6\rangle} * c + p\ _{\langle 5\rangle} * g_{\uparrow} + l$$
$$Cost_{Worker} = O(n_{worker})\ _{\langle 8\rangle} * c$$

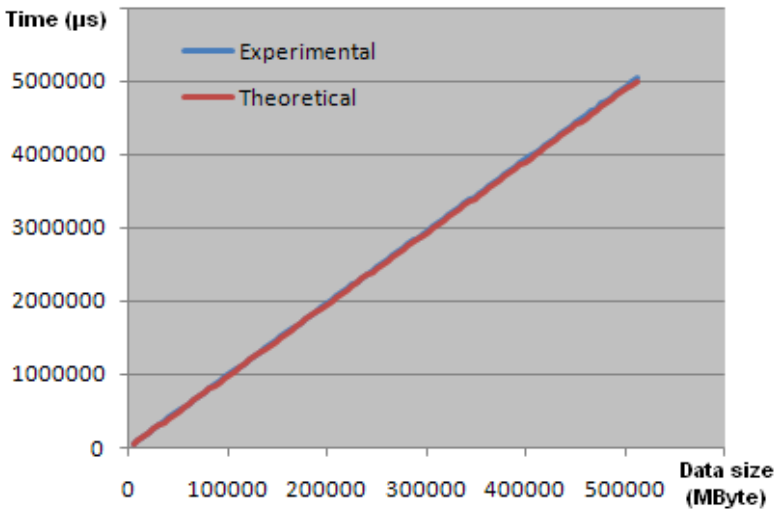



Figure 2: Reduction (average relative error: 1.17%)

The measurements show a near perfect match of measured performance with predicted performance (obtained from the cost formula with machine parameters measured independently of this algorithm).

### 5.2.2 Parallel Scan

We perform this algorithm in 2 steps:
1. Each *worker* performs a sequential scan. Then, the *master* fetches the last computed element from each *child*. After that, it performs a sequential scan of the fetched data.
2. *Master* scatters the result to its *children*. Then, each *child* computes the sum of the received value and its computed data. Finally, the *children* obtain the final result.

The cost of the super-step (subscripts $\langle\rangle$ refer to the pseudo-code line):

$$
\begin{aligned}
Cost_{Master} = & \max_{i=1..p}(Step1_{Child_i\ \langle 1\text{-}3\rangle} + O(1)\ _{\langle 1\text{-}4\rangle} * c_i) \\
& + \max_{i=1..p}(Step2_{Child_i\ \langle 2\text{-}5\rangle} + O(n_{child_i})\ _{\langle 2\text{-}4\rangle} * c_i) \\
& + (O(p)\ _{\langle 1\text{-}7\rangle} + O(p-1)\ _{\langle 1\text{-}8\rangle}) * c \\
& + p\ _{\langle 1\text{-}6\rangle} * g_{\uparrow} + p\ _{\langle 2\text{-}2\rangle} * g_{\downarrow} + 2 * l \\
Cost_{Worker} = & O(n_{worker})\ _{\langle 1\text{-}10\rangle} * c
\end{aligned}
$$

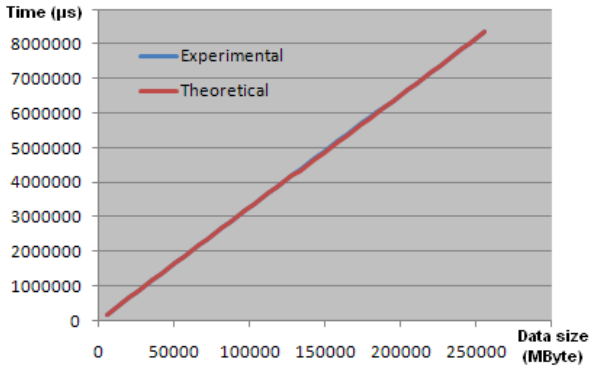



Figure 3: Scan (average relative error: 0.43%)

Again the pseudo-code and measurements strongly support our claims that SGL is simple to use and that its performance model is reliable. We now implement a BSP parallel sorting algorithm in SGL.

### 5.2.3 Parallel Sorting

This algorithm is based on Parallel Sorting by Regular Sampling(PSRS)[SS92] [5], a *partition-based* algorithm. We implement this algorithm in 5 steps:
1. Each *worker* performs a local sort and selects $P$ (number of workers) samples which are gathered onto *root-master*.
2. *Root-master* performs a local sort of $P^2$ gathered samples. Then, it picks $P$-1 almost-equally spaced pivots from the sorted samples.
3. *Root-master* broadcasts these pivots to all *workers*. After that, each *worker* produces $P$ partitions[6] of its local data using the $P$-1 pivots.
4. *Master* gathers the partitions which are not already in place.
5. *Master* scatters the gathered partitions to its *children* according to partitions' index. Then, each *worker* performs a local merge of the received partitions.

[Suj96] showed that the computational cost of this algorithm is $2\frac{n}{p}(\log n - \log p + \frac{p^3}{n}\log p)$, and the communication cost is $g\frac{1}{p}(p^2(p-1)+n)+4l$ in BSP. Thus, the total cost of this algorithm

[5] We could use the algorithm of [HJB98] to improve the performance.
[6] Partition $i$ contains values that should now move to worker $i$.

**Algorithm 2** Parallel Scan

Scan(IN $\overrightarrow{src}$, OUT $\overrightarrow{res}$)
**begin**
  Step1($\overrightarrow{src}$, $\overrightarrow{mid}$);
  Step2($\overrightarrow{mid}$, $\overrightarrow{res}$);
**end**

Step1(IN $\overrightarrow{src}$, OUT $\overrightarrow{res}$)
**begin**
1: **if master then**
2:  **par do**
3:  Step1($\overrightarrow{src}$, $\overrightarrow{res}$); $\{\Longleftarrow Step1_{child_i}\}$
4:  $x := \overrightarrow{res}$[**len** $\overrightarrow{res}$]; $\{\Longleftarrow O(1)_{child_i}\}$
5:  **end par**
6:  **gather** $x$ **to** $\overrightarrow{res}$; $\{\Longleftarrow p * g_{\uparrow} + l\}$
7:  ShiftRight($\overrightarrow{res}$); $\{\Longleftarrow O(p)\}$
8:  $\overrightarrow{res}$ := LocalScan $\overrightarrow{res}$; $\{\Longleftarrow O(p)\}$
9: **else**
10:  $\overrightarrow{res}$ := LocalScan $\overrightarrow{src}$; $\{\Longleftarrow O(n)\}$
11: **end if**
**end**

Step2(IN $\overrightarrow{src}$, OUT $\overrightarrow{res}$)
**begin**
1: **if master then**
2:  **scatter** $\overrightarrow{src}$ **to** $x$; $\{\Longleftarrow p * g_{\downarrow} + l\}$
3:  **par do**
4:  $\overrightarrow{res} := \overrightarrow{src} + x$; $\{\Longleftarrow O(n_{child_i})\}$
5:  Step2($\overrightarrow{res}$, $\overrightarrow{res}$); $\{\Longleftarrow Step2_{child_i}\}$
6:  **end par**
7: **else**
8:  **skip**;
9: **end if**
**end**

LocalScan(IN $\overrightarrow{src}$, OUT $\overrightarrow{res}$)
**begin**
  $res[1] := src[1]$;
  **for** $i$ **from** 2 **to** (**len** $src$) **do**
  $res[i] := res[i-1] + src[i]$;
  **end for**
**end**

ShiftRight(INOUT $\overrightarrow{lst}$)
**begin**
  **for** $i$ **from** (**len** $\overrightarrow{lst}$) **to** 2 **do**
  $\overrightarrow{lst}[i] := \overrightarrow{lst}[i-1]$;
  **end for**
  $\overrightarrow{lst}[1] := 0$;
**end**

**Algorithm 3** Parallel Sorting by Regular Sampling (Part I)

Step0

1: initialize $pid := 1$ in each *worker*
2: use parallel scan (c.f. Section 5.2.2) to computer the *pid* for each each *worker*
3: set $lowerPid$, $upperPid$ and $maxPid$ in *masters*

Step1(IN $\overrightarrow{arr}$, OUT $\overrightarrow{sam}$)
**begin**
1: **if master then**
2: **par do**
3: Step1($\overrightarrow{arr}$, $\overrightarrow{sam}$);
4: **end par**
5: **gather** $\overrightarrow{sam}$ **to** $\widetilde{tmp}$;
6: $\overrightarrow{sam}$ := Concatenate($\widetilde{tmp}$);
7: **else**
8: QuickSort($\overrightarrow{arr}$);
9: SelectSamples($\overrightarrow{arr}$, $\overrightarrow{sam}$);
10: **end if**
**end**

Step2(IN $\overrightarrow{sam}$, OUT $\overrightarrow{pvt}$)
**begin**
1: QuickSort($\overrightarrow{sam}$)
2: PickPrivots($\overrightarrow{sam}$, $\overrightarrow{pvt}$);
**end**

Step3(IN $\overrightarrow{arr}$, IN $\overrightarrow{pvt}$, OUT $\widetilde{blk}$)
**begin**
1: **if master then**
2: **for** $i$ **from** 1 **to** $numChd$ **do**
3: $\widetilde{tmp}[i] := \overrightarrow{pvt}$;
4: **end for**
5: **scatter** $\widetilde{tmp}$ **to** $\overrightarrow{pvt}$;
6: **par do**
7: Step3($\overrightarrow{arr}$, $\overrightarrow{pvt}$, $\widetilde{blk}$);
8: **end par**
9: **else**
10: BuildPartitions($\overrightarrow{arr}$, $\overrightarrow{pvt}$, $\widetilde{blk}$);
11: **end if**
**end**

**Algorithm 4** Parallel Sorting by Regular Sampling (Part II)

Step4(IN $\widetilde{blk}$, OUT $\widetilde{stay}$, OUT $\widetilde{move}$)
**begin**
1: **if master then**
2: **par do**
3: Step4($\widetilde{blk}$, $\widetilde{stay}$, $\widetilde{move}$);
4: **end par**
5: **for** $i$ **from** 1 **to** $maxPid$ **do**
6: **gather** $\widetilde{move}[i]$ **to** $\widetilde{tmp}$;
7: **if** $i < lowerPidori > upperPid$ **then**
8: $\widetilde{move}[i])$ := Concatenate($\widetilde{tmp}$);
9: **else**
10: $\widetilde{stay}[i]$ := Concatenate($\widetilde{tmp}$);
11: **end if**
12: **end for**
13: **else**
14: **for** $i$ **from** 1 **to** $maxPid$ **do**
15: **if** $i = pid$ **then**
16: $\widetilde{stay}[pid]$ := $\widetilde{blk}[pid]$;
17: **else**
18: $\widetilde{move}[i]$ := $\widetilde{blk}[i]$;
19: **end if**
20: **end for**
21: **end if**
**end**

Step5(IN $\widetilde{stay}$, IN $\widetilde{move}$, OUT $\widetilde{\overrightarrow{arr}}$)
**begin**
1: **if master then**
2: **scatter** Bundle($\widetilde{move}$) **to** $\widetilde{\overrightarrow{arr}}$;
3: **par do**
4: $\widetilde{tmp}$ := Unbundle($\widetilde{\overrightarrow{arr}}$);
5: **for** $i$ **from** $lowerPid$ **to** $upperPid$ **do**
6: $\widetilde{move}[i]$ := Concatenate($\widetilde{tmp}[i]$, $\widetilde{stay}[i]$);
7: **end for**
8: Step5($\widetilde{stay}$, $\widetilde{move}$, $\widetilde{\overrightarrow{arr}}$);
9: **end par**
10: **else**
11: $\overrightarrow{arr}$ := MergeSort($\widetilde{move}[pid]$);
12: **end if**
**end**

implemented in SGL is

$$2\frac{n}{p}(\log n - \log p + \frac{p^3}{n}\log p) * c + (p^2(p-1)+n) * G + 4 * L$$

where $G$ is the sum of all $g$ from each level and $L$ is the sum of all $l$ from each level.

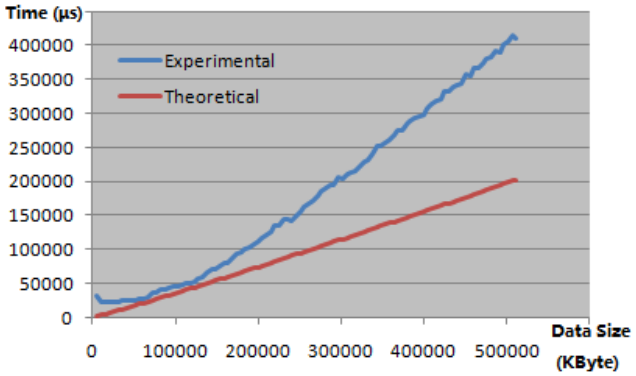



Figure 4: Parallel Sorting by Regular Sampling

Our performance prediction is based on $SeqSort(n) = O(n \log n)$ and it remains to be improved.

The pseudo-code shows that a BSP algorithm is easily implemented in SGL. We now provide some quantification of SGL's capacity to generate large speed-ups.

### 5.3 Speed-up and Efficiency

We tested speed-up and efficiency using the scan algorithm of section 5.2.2. During the measurements, the size of input data is fixed at 100 MBytes. The formulas we used are:

$$Speedup = \frac{ExecutionTime_{numproc=16}}{ExecutionTime_{numproc=16...128}}$$

$$Efficiency = \frac{Speedup}{numproc/16}$$

First of all, we fixed the number of cores at 8 for each node and varied the number of nodes from 2 to 16. We obtained the upper half of the table. After that, we fixed the number of node at 16 and varied the number of core of each node from 1 to 8. We obtain the lower half of the table.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Efficiency | 1 | 0.995 | 0.991 | 0.987 | 0.982 | 0.978 | 0.974 | 0.969 |
| Speed-up | 1 | 1.99 | 2.97 | 3.95 | 4.91 | 5.87 | 6.82 | 7.75 |
| Node scale-out | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Num of proc | 16 | 32 | 48 | 64 | 80 | 96 | 112 | 128 |
| Core scale-out | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Speed-up | 1 | 1.99 | 2.97 | 3.95 | 4.91 | 5.87 | 6.82 | 7.75 |
| Efficiency | 1 | 0.995 | 0.991 | 0.987 | 0.982 | 0.978 | 0.974 | 0.969 |

We measured very small differences between two types of scale-out experiments but they are not visible at the table's precision.

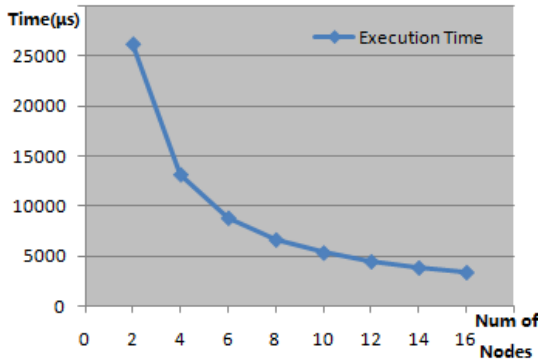



Figure 5: Node-level scale-out (8 cores per node)

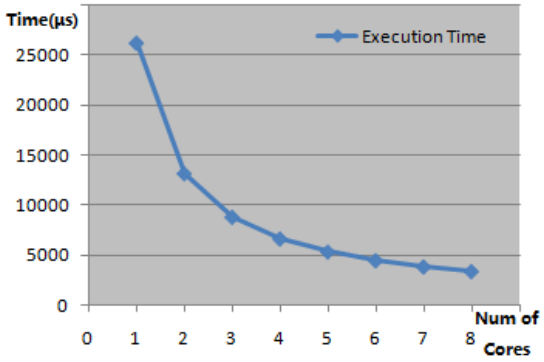



Figure 6: Core-level scale-out (16 nodes)

# 6 Conclusion

Scatter-gather parallel operations are old, well-understood concepts and we do not pretend that SGL's use of them is novel. What this report has attempted to motivate and support is 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 centralized communications. From the perspective of programming research, SGL is a proposal to 1. adapt BSML to a hierarchical machine structure 2. remove its general communication `put` primitive[7] 3. replace `mkpar` with the scatter operation and 4. replace `proj` with the gather operation.

Our initial experiments with language definition, programming and performance measurement show that SGL combines a clean semantics, simplified programming of BSP-like algorithms and dependable performance measurement. One remaining open problem is the implicit treatment of horizontal communication as is needed for operations like sample-sort or bucket-sort [SS92]. But we are confident that the large body of knowledge on algorithms for cost-explicit models will yield many solutions to this problem. It is our opinion that SGL's lesser expressive power with respect to message-passing structures is not only necessary for software reliability but also coherent with one of the most basic concept in computing, namely recursive structures.

# 7 Future Work

Ongoing work around SGL aims at developing 1. an MPI/OpenMP implementation of a library for SGL's parallel operations 2. a compiler for the simple imperative SGL language as described in this report 3. performance prediction for this compiler based on our performance model

[7] `Put` is no more a primitive but remains a possible implementation tool.

4. optimizations based on the performance prediction model and [BDH+09] the fundamental equation of modelling $T_{total} = T_{comp} + T_{comm} - T_{overlap}$
5. a compiler from EXQIM's domain-specific source application language EXQIL (which has no explicit parallel construct) to SGL.

All of the above aspects of SGL's development assume that it can be compiled-executed with a variable SGL-machine view to more or less match the physical architecture. For example a network of bi-processors built from quadri-core processors can have one, two or three levels when viewed as an SGL computer.

Future work will investigate 1. horizontal child-to-child communications as implicit semantics or optimizations for SGL program executions, 2. horizontal and vertical load balancing for heterogeneous machines, 3. pipelining or overlap behaviour in the operational semantics or as optimizations, 4. SGL support for algorithms that use such horizontal communications, 5. including memory size in the model, 6. extended SGL implementation with accelerators (CUDA/Cell/SSE/AVX/etc.), 7. extended SGL implementation to supporting fault-tolerance.

## 8 Acknowledgement

The first author thanks EXQIM S.A.S for an industrial doctoral scholarship under the French government's CIFRE scheme. The second author thanks Bill McColl for introducing him to the BSP model in the early 1990's. SGL is inspired by the work of colleagues who are too numerous to cite here, but special thanks are due to Frédéric Loulergue et al. for their work on BSML. Both authors thank Frédéric Gava for useful comments on this report and EXQIM for access to the SGI hardware on which we have conducted our experiments.

## References

[ABB+03] George Almasi, Ralph Bellofatto, Jose Brunheroto, Calin Cascaval, Jose G. Castanos, José G, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moreira, and Alda Sanomiya. An overview of the Blue Gene/L system software organization. In *In Proceedings of Euro-Par 2003 Conference, Lecture Notes in Computer Science*, pages 543–555. Springer-Verlag, 2003.

[Adv09] Sarita V. Adve. Memory models: a case for rethinking parallel languages and hardware. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 45–45, New York, NY, USA, 2009. ACM.

[AISS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. Technical report, Santa Barbara, CA, USA, 1995.

[BDH+08] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press.

[BDH+09] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and Jose Carlos Sancho. Using performance modeling to design large-scale systems. *Computer*, 42:42–49, 2009.

[BJvOR03] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Comput.*, 29:187–207, February 2003.

[BP04] J.L. Bosque and L.P. Perez. HLogGP: a new parallel computational model for heterogeneous clusters. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 403 – 410, 2004.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28:1–12, July 1993.

[Col89] M. Cole. *Algorithmic Skeletons, Structural Management of Parallel Computation.* MIT Press, 1989.

[CZZZ09] WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. LogGPO: An accurate communication model for performance prediction of mpi programs. *Science in China Series F: Information Sciences*, 52:1785–1791, 2009. 10.1007/s11432-009-0161-2.

[DFRC93] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of the ninth annual symposium on Computational geometry*, SCG '93, pages 298–307, New York, NY, USA, 1993. ACM.

[GL05] Frédéric Gava and Frédéric Loulergue. A functional language for departmental metacomputing. *Parallel Processing Letters*, 15(3):289–304, 2005.

[GR88] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[Hai98] G. Hains. Subset synchronization in BSP computing. In H. R. Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998. CSREA Press.

[HF93] G. Hains and C. Foisy. The data-parallel categorical abstract machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 56–67, Munich, June 1993. Springer.

[HJB98] David R. Helman, Joseph JáJá, and David A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *J. Exp. Algorithmics*, 3, September 1998.

[JB07] C. R. Johns and D. A. Brokenshire. Introduction to the Cell broadband engine architecture. *IBM J. Res. Dev.*, 51:503–519, September 2007.

[joRs08] IBM journal of Research and Development staff. Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52:199–220, January 2008.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005.

[KTJR05] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38:32–38, 2005.

[LGAD04] Frédéric Loulergue, Frédéric Gava, M. Arapinis, and Frédéric Dabrowski. Semantics and implementation of minimally synchronous parallel ML. *International Journal of Computer and Information Science*, 5(3):182–199, 2004.

[LHF00] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[Lou00] F. Loulergue. *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.

[Lou10] F. Loulergue. Bulk synchronous parallel ML, 2010. http://bsmllib.free.fr.

[MW98] William F. McColl and David Walker. Theory and algorithms for parallel computation. In David J. Pritchard and Jeff Reeve, editors, *Euro-Par*, volume 1470 of *Lecture Notes in Computer Science*, pages 863–864. Springer, 1998.

[SMD+10] A.C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via multithreaded and multicore CPUs. *Computer*, 43(3):24 –32, 2010.

[SS92] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[Suj96] Ronald Sujithan. Bsp parallel sorting by regular sampling - algorithm and implementation. Technical report, Computing Laboratory, University of Oxford, 1996.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[Val08] Leslie 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.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.