Efficient Compilation of High-Level Data Parallel Algorithms

Dan Suciu Val Tannen*

Deptartment of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389, USA
Email: (suciu,val)@saul.cis.upenn.edu

Abstract

We present a high-level parallel calculus for nested sequences, $\mathcal{NSC}$, offered as a possible theoretical "core" of an entire class of collection-oriented parallel languages. $\mathcal{NSC}$ is based on *while*-loops as opposed to general recursion. A formal, machine independent definition of the parallel time complexity and the work complexity of programs in $\mathcal{NSC}$ is given. Our main results are: (1) We give a translation method for a particular form of recursion, called *map*-recursion, into $\mathcal{NSC}$, that preserves the time complexity and adds an arbitrarily small overhead to the work complexity, and (2) We give a compilation method for $\mathcal{NSC}$ into a very simple vector parallel machine, which preserves the time complexity and again adds an arbitrarily small overhead to the work complexity.

1 Introduction

There are many advantages to programming in a high-level language. However, while sequential algorithms are most of the time designed and evaluated in reasonably high-level terms, the situation with parallel algorithms is - by necessity, so far - more complicated. The issue is intimately connected with the existing efforts to bridge the gap between the theoretical design of parallel algorithms and practical programming on massively parallel computers.

In the case of *data parallelism*, the work of Blelloch [Ble90, Ble93] and Blelloch and Sabot [BS90] has made substantial progress on this issue. For example, if we manage to represent an algorithm in a high-level language such as NESL with a certain work and time (a.k.a. element or step) complexity and if the representation satisfies certain restrictions then we are guaranteed an implementation of the same algorithm with the same asymptotic time and work complexity in terms of a low-level parallel vector model, which in turn admits efficient implementations on various architectures, for example the CM2. The present paper is proposing a different treatment of similar goals.

*The authors were partially supported by NSF Grant CCR-90-57570

We start with a somewhat abstract high-level language which represents and manipulates mostly nested sequences (lists) and so we called it $\mathcal{NSC}$, for *nested sequence calculus* (section 3). We regard $\mathcal{NSC}$ as a possible theoretical "core" of an entire class of collection-oriented parallel languages. In keeping with the tenets of data parallelism [HS86], $\mathcal{NSC}$'s only parallel operation is *map* (apply-to-all). We give a precise high-level definition of parallel complexity (in the work and time framework [Jaj92]) for $\mathcal{NSC}$ programs.

Blelloch [Ble90, Ble93] gives convincing evidence that nested *map*'s on nested sequences (what he calls *nested parallelism*) can enhance the expressiveness of a data parallel language. But these high-level features are quite removed from concrete parallel architectures or even the parallel vector model and need to be compiled away. *Unnesting* the nested parallelism is at the center of the compilation technique of [Ble90, BS90, Ble93]. However, in a language with general recursion, this technique is guaranteed to preserve the asymptotic parallel complexity only for programs that satisfy a certain semantic condition called *containement*.

$\mathcal{NSC}$ is based on *while*-loops rather than general recursion. This will surely impose some limitations, although not that many: our *first main result* consists of showing that a large and practically relevant class of programs, called *map*-recursive, can be translated into $\mathcal{NSC}$ while asymptotically preserving the time complexity and adding an arbitrarily small overhead to the work complexity (theorem 4.2). It even turns out that some recursive programs which are not contained in the sense of [Ble90] are in fact *map*-recursive. The major benefit however is that we can compile $\mathcal{NSC}$ without the need for an unbounded stack of vectors, as general recursion would require. Avoiding the stack is a good idea because SIMD architectures associate a relatively small memory with each processor. A program that generates many entries in its vector stack will run out of memory even if the vectors are very short and hence much of the *total* amount of memory of the machine remains unused. We believe that our compilation technique can lead to better memory management. Of course, this needs to be tested in practice.

Following Blelloch, we define a simple parallel vector model in order to describe abstractly the class of target architectures for our compilation method (section 2). Our BVRAM (Bounded Vector Random Access Machine) differs from the VRAM [Ble90] primarily in that it has a finite number of vector registers. This emphasizes the absence of a run-time vector stack. Of course the number of registers needed depends on the source program being compiled. Another important difference is that we need less powerful communication primitives. The BVRAM has no general permutation instruction, and its communication primitives can be implemented on a butterfly network with $n \log n$ nodes in $O(\log n)$ steps. The BVRAM can be efficiently implemented on SIMD architectures such as CM2 and MasPar MP-1, and it has the potential of efficient implementation on MIMD machines as well, such as CM5, Paragon XP/S, KSR1 etc.

Our *second main result* is a technique that compiles any $\mathcal{NSC}$ program into a BVRAM program again while asymptotically preserving the time complexity and adding an arbitrarily small overhead to the work complexity (theorem 7.1). Along the way we also give a simulation that allows us to understand $\mathcal{NSC}$ complexity in terms of the complexity of computations on a certain flavor of PRAM (proposition 3.2), we show how to implement the BVRAM instructions on a butterfly network (proposition 2.1), we connect $\mathcal{NSC}$ with some standard parallel complexity classes (proposition 6.2), we show how to represent in $\mathcal{NSC}$ Valiant's $O(\log n \log\log n)$ time sorting algorithm [Val75, Jaj92] (section 5), and, as part of the compilation process, we define an intermediate abstract language - the *sequence algebra* - which has the same power as BVRAM's but may prove more flexible in connecting to the designs of the future (section 7).

$\mathcal{NSC}$ borrows heavily from our experience with languages for collection types [BTS91, BBW92] and it is worthwhile mentioning that many of its operations make as much sense for sets and bags (multisets) as for lists (sequences). It matters to us, though it may not be so relevant to the goals of this paper, that $\mathcal{NSC}$ is based on a clear, statically checkable type system, that we understand the meaning of $\mathcal{NSC}$ programs independently of their parallel execution, and that we know how to reason about them - for example how to validate source to source optimizations. We have in mind applications to databases and this naturally brings up important complexity issues. In a previous paper we have shown a tight connection between a related data parallel language for sets and the class NC [SBT94]. This in turn has led us to the more practical questions addressed here.

2 The Target: Bounded Vector Random Access Machines

To compile the higher level programming language described in section 3 only a very simple vector parallel model is needed. The Bounded Vector Random Access Machine, BVRAM, is a restriction of the VRAM introduced in [Ble90], in that it only admits a fixed number of registers, and has only particular communication primitives, not a general permutation. The BVRAM can be efficiently implemented on a wide range of parallel architectures, because: (1) only a simple, rather particular form of communication is needed to implement every instruction of the BVRAM, and (2) memory management at each processor is simplified by having only a bounded number of vector registers, as opposed to an unbounded number in the VRAM model.

A BVRAM, M, consists of a fixed number of **vector registers** $V_1, \ldots, V_r$. Each V_i can hold a sequence (a vector) of natural numbers of arbitrary, but finite length. To keep the model simple, we don't include *scalar* registers: a number is represented by a sequence of length 1. A *program* for M is a sequence of labeled instructions, from the following instruction set. For some of the instructions below, it is convenient to view a pair of registers V_i, V_j in which the length of the first equals the sum of the numbers in the second as a *nested sequence*. E.g., intuitively we view $[x_0, x_1, z_0, z_1, z_2], [2, 0, 3]$ as standing for the nested sequence $[[x_0, x_1], [], [z_0, z_1, z_2]]$.

- Move instruction: $V_i \leftarrow V_j$.
- Arithmetic operations, of the form $V_i \leftarrow V_j \ op \ V_k$. Here *op* is an arithmetic operation from a set Σ. V_j and V_k must be arrays of the same length, and the operation *op* is applied simultaneously on all all elements of V_j and V_k from the same positions, and the result is stored in V_i. In general we leave Σ unspecified, but mention here that for theorems 4.2 and 7.1 Σ has to contain $+, \dot{-}, *, /, \textit{right-shift}, \log_2$, while for proposition 6.2 we require that all operations in Σ be in NC. *Monus*, written $m \dot{-} n$, is defined as $m - n$ when $m \geq n$ and 0 otherwise.
- Sequence oriented operations: $V_i \leftarrow []$ loads the empty sequence in V_i. $V_i \leftarrow [n]$, where $n \in \mathbb{N}$ loads the singleton sequence $[n]$ into V_i. $V_i \leftarrow V_j @ V_k$ appends V_j and V_k and stores the result in V_i. $V_i \leftarrow [\mathit{length}(V_j)]$ computes the length of V_j. $V_i \leftarrow \mathit{enumerate}(V_j)$ loads the sequence $[0, 1, \ldots, n-1]$ into V_i, where n is the length of V_j.
- Bounded monotone routing $V_i \leftarrow \mathit{bm\_route}(V_j, V_k, V_l)$; here V_k and V_l must have the same length. The effect is that each element in V_l is replicated a number of times equal to the corresponding number in V_k. In addition, it is required that the result matches in length the sequence V_j (i.e. initially V_j, V_k represent a nested sequence). E.g. if $V_j = [x_0, x_1, z_0, z_1, z_2]$, $V_k = [2, 0, 3]$ and $V_l = [a, b, c]$, then the instruction $V_i \leftarrow \mathit{bm\_route}(V_j, V_k, V_l)$ stores $[a, a, c, c, c]$ into V_i.
- Segmented bounded monotone routing $V_i \leftarrow \mathit{sbm\_route}$ (V_j, V_k, V_l, V_m). Here, V_j, V_k and V_l, V_m must be nested sequences, and $\mathit{length}(V_k) = \mathit{length}(V_m)$. Then, the subsequences of V_l are replicated according to the numbers in V_k and the result is stored in V_i. E.g., suppose $V_j = [x_0, x_1, z_0, z_1, z_2]$, $V_k = [2, 0, 3]$, $V_l = [a_0, a_1, b_0, b_1, b_2, c_0, c_1, c_2]$ and $V_m = [2, 3, 3]$. Then, after $V_i \leftarrow \mathit{sbm\_route}(V_j, V_k, V_l, V_m)$, V_i will hold the value $[a_0, a_1, a_0, a_1, c_0, c_1, c_2, c_0, c_1, c_2, c_0, c_1, c_2]$. In the particular case when V_k, V_m have length one, this computes the *cartesian product* of V_j and V_l. Note that the length of the output is $\leq \mathit{length}(V_j) * \mathit{length}(V_l)$ and that *bm_route* can be expressed with two *sbm_route* instructions.
- Selection $V_i \leftarrow \sigma(V_j)$. The effect is that the nonzero values of V_j are packed and moved into V_i. E.g. if $V_j = [3, 0, 1, 0, 0, 4]$, then $[3, 1, 4]$ is stored in V_i.
- The unconditional jump goto l and the conditional jump if $\mathit{empty?}(V_i)$ then goto l, where l is a label of some instruction. The conditional jump is taken iff V_i currently holds the empty sequence.
- *halt*, stops the program.

We associate with each BVRAM program P two numbers: r_i, r_o, the number of *input* and *output* registers. P expects r_i inputs in the registers $V_1, \ldots, V_{r_i}$, and returns r_o outputs, in $V_1, \ldots, V_{r_o}$. For some input, the result of P

might be undefined, if P enters an infinite loop, or if an error occurs. For a terminating execution of P, we define the **parallel time complexity** T to be the total number of instruction executed by P, i.e. each instruction is considered to have parallel time complexity 1. Similarly, we define the **work complexity** W as the sum of the work complexities of all instructions executed by P, where the work complexity of some instruction is defined to be the sum of the lengths of its input and output registers.

As opposed to VRAMs [Ble90] there is no general permutation instruction on a BVRAM (but one can be computed with an increase in the time or work complexity). This may lead to more efficient implementations on fixed-connection networks, as exemplified by the following proposition.

Proposition 2.1 *Any BVRAM instruction of work complexity W can be implemented in time $O(\log n)$ on a butterfly network with $n \log n$ nodes, where $n = O(W)$, using only* oblivious *routing algorithms.*

Proof. (Sketch) The arithmetic operations involve no communication at all, thus can be implemented in $O(1)$ steps. The append operation $V_i \leftarrow V_j @ V_k$ only requires a *monotone routing* of the values in V_k. This can be done in $O(\log n)$ steps, using the greedy routing algorithm, see [Lei92], pp. 534. *bm_route* is implemented by a monotone routing, and takes $O(\log n)$ steps with the greedy algorithm. For *sbm_route*, suppose first that $length(V_j) = length(V_l) = 1$, i.e. *sbm_route* computes the cartesian product of V_i and V_k. Also, suppose that the length of V_i and V_k are powers of 2, namely 2^p and 2^q respectively. Take $n = 2^{p+q}$; then we have 2^p packets residing in the first 2^p rows of a butterfly with 2^{p+q} rows, and we have to route the packet with address $00\ldots 0u_{p-1}\ldots u_1u_0$ to all addresses of the form $v_{q-1}\ldots v_1v_0u_{p-1}\ldots u_1u_0$. This is done in q stages, starting with the higher dimension, using the greedy algorithm. In the general case of *sbm_route*, we have to replicate a number of smaller sequences. First, round upwards to the closest power of 2 the length of each such subsequence, and spread the sequences such that each sequence of lenght m starts at an address divisible by m. Next, perform in parallel all replications, as described above. 2

When the number n of available processors is less than the number W of elements in an array, then we group $\frac{W}{n}$ adjacent elements of the array in the same processor. The above proposition can be extended to this case: some instruction of complexity W can be implemented on a butterfly network in $O(\frac{W}{n}\log n)$ steps.

3 The Source: The Nested Sequence Calculus ($\mathcal{NSC}$)

We use *types* to explain the structure of $\mathcal{NSC}$ and classify its features. The types are given by the grammar $t ::= unit \mid \mathbb{N} \mid t \times t \mid t + t \mid [t]$. *unit* has exactly one value: the empty tuple (). $\mathbb{N}$ is the type of nonnegative integers. The values of the **product type** $s \times t$ are pairs (x, y), with $x \in s, y \in t$. $[t]$ is the **finite sequences type** over t: it contains all sequences $[x_0, \ldots, x_{n-1}]$, with $n \geq 0$ and $x_0, \ldots, x_{n-1} \in t$. $s + t$ is the **disjoint union type** of s and t; its values are of the form $in_1(x)$ with $x \in s$ and $in_2(y)$ with $y \in t$. We define the **boolean** type $\mathbb{B} \stackrel{\text{def}}{=} unit + unit$, and identify its values $in_1(())$ and $in_2(())$ with *true* and *false* respectively. Extending the list of built-in types with reals, strings, etc., can be done while preserving all results.

The primitives of $\mathcal{NSC}$ are chosen to be operations naturally associated to its types. Its expressions belong to one of two distinct syntactic categories: **terms**, denoted by M, N, P, U, V, etc., which have some type t, and **functions**, denoted by F, G, etc., have associated two types, the domain s and codomain t. By abuse of the language we say in this case that the "type" of some function F is $s \to t$. However $s \to t$ is not a type per se, which makes constructs like $s \to (t_1 \to t_2)$ or $(s_1 \to s_2) \to t$ impossible. Both terms and functions may contain free variables. See [ST94] for a full and formal description of the language):

- Variables x, error Ω, constants n (where $n \in \mathbb{N}$), arithmetic operations $M\ op\ N$, where $op \in \Sigma$ (recall from section 2 that $\Sigma = \{+, \dot{-}, *, /, \ldots\}$), and equality $M = N$.
- Constructs associated with the product type: $()$, π_1, π_2, (M, N). Here () denotes the empty tuple, (M, N) is a pair, while π_1, π_2 are the projections, with the meaning $\pi_1(x, y) \stackrel{\text{def}}{=} x, \pi_2(x, y) \stackrel{\text{def}}{=} y$.
- Constructs associated with the sum type: $in_1(M)$, $in_2(N)$, and *case* M *of* $in_1(x) \Rightarrow N\ in_2(x) \Rightarrow P$. The latter is defined to be equal to $N[U/x]$ when $M = in_1(U)$, and respectively to $P[V/y]$, when $M = in_2(V)$.
- Constructs associated with functions: $\lambda x : s.M$ and $F(M)$. The former is called a *lambda abstraction*, and is a *function* (as opposed to a *term*), of type $s \to t$, provided that M is a term of type t. The second construct, $F(M)$, is a term called *function application* having type t, provided that F is some function of type $s \to t$, and M is a term of type s. Although the type s is part of the syntax of $\lambda x : s.M$, we shall drop it when it is clear from the context. Note that $\lambda x : s.F$, where F is a function, is not a legal construct in $\mathcal{NSC}$, nor is $\lambda x : s \to t.M$, i.e. no higher order functions are allowed.
- Iteration: $while(P, F)$ is some function of type $t \to t$, provided that P and F are functions of type $t \to \mathbb{B}$ and $t \to t$ respectively.
- Constructs associated with collections (these constructs work on sequences but also make sense for other kinds of collections, like sets and bags [BBW92]): [], $[M]$, $M@N$, $flatten(M)$, $length(M)$, $get(M)$, and $map(F)$. Here [] denotes the empty sequence, $[M]$ is the singleton sequence, and @ is the append operator. Next, $flatten\ ([x_0, \ldots, x_{n-1}]) \stackrel{\text{def}}{=} x_0 @ x_1 @ \ldots @ x_{n-1}$, and $length(M)$ returns the length of some sequence. *get* is defined by $get([x]) \stackrel{\text{def}}{=} x$, $get([]) \stackrel{\text{def}}{=} get([x_0, x_1, \ldots]) \stackrel{\text{def}}{=} \Omega$. Finally $map(F)$ is a function of type $[s] \to [t]$, provided that F is a function of type $s \to t$. Its meaning is: $map(F)([x_0, \ldots, x_{n-1}]) \stackrel{\text{def}}{=} [F(x_0), \ldots, F(x_{n-1})]$.
- Constructs associated only to sequences, and not to other kinds of collections: $zip(M, N)$, $enumerate(M)$, and $split(M, N)$. The meanings are: $zip([x_0, \ldots, x_{n-1}], [y_0, \ldots, y_{n-1}]) \stackrel{\text{def}}{=} [(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})]$ (*zip* is undefined if its two arguments have different lengths), $enumerate([x_0, \ldots, x_{n-1}]) \stackrel{\text{def}}{=} [0, \ldots, n-1]$. Finally

split(M, N) splits M according to the numbers contained in N; e.g. $split([a,b,c,d,e,f],[3,0,1,0,2]) \stackrel{\text{def}}{=} [[a,b,c],[],[d],[],[e,f]]$. It is defined only if the sum of elements in N equals the length of M.

Note that any function in $\mathcal{NSC}$ can, in a fixed amount of time, only increase the size of its input by some polynomial. Had we introduced as a primitive in the language something like $\iota(n) \stackrel{\text{def}}{=} [0,\ldots,n-1]$, which generates an arbitrarily long list out of a number, this property would fail. From this small set of primitives, we can *derive* a rich set of functions. Some examples:

Database projections. $\Pi_i : [t_1 \times t_2] \to [t_i]$, $\Pi_i \stackrel{\text{def}}{=} map(\pi_i)$.

Conditionals. if $x = y$ then M else N is expressed by $case\ (x = y)\ of\ in_1(u) \Rightarrow M\ in_2(v) \Rightarrow N$, where u, v are variables of type *unit*, not occurring in M, N.

Broadcasting. $\rho_2(x, [y_0,\ldots,y_{n-1}])$, which is defined to be $[(x,y_0),\ldots,(x,y_{n-1})]$, can be expressed as $\rho_2(x,y) \stackrel{\text{def}}{=} map(\lambda(v).(x,v))(y)$ and has the type $\rho_2 : s \times [t] \to [s \times t]$. When x itself is a sequence, $\rho_2(x,y)$ essentially computes the cartesian product of x and y. (The name ρ_2 is motivated by other considerations [BBW92].)

Bounded monotone routing. $bm\_route((u,d),x)$, expressed as $\Pi_1(flatten(map(\rho_2)(zip(x, split(u,d)))))$, has type $bm\_route : ([s] \times [\mathbb{N}]) \times [t] \to [t]$. *bm_route* is essentially the same operation as secribed in section 2. E.g. $bm\_route(([u_0, u_1, u_2, v_0, v_1],[3,0,2]),[a,b,c]) = [a,a,a,c,c]$. The bound u prohibits us from constructing a very long sequence in constant parallel time. An *unbounded* monotone routing $m\_route : [\mathbb{N}] \times [t] \to [t]$ can be defined in $\mathcal{NSC}$ (with *while*), but requires more than a constant number of parallel steps. This is indeed necessary, since $m\_route([n],[a])$, produces the sequence $[a,a,\ldots,a]$ of lenght n, whose size is not polynomially bounded by the input. Finally, note that in the context of nested sequences, our bounded monontone routing is not truly "monotone". Indeed, $bm\_route(([(),()],[2]), [[a,b,c]]) = [[a,b,c],[a,b,c]]$, and the relative order of a, b and c has not been preserved. This forces us to introduce *sbm_route* in the BVRAM model.

Selections. $\sigma_1 : [s+t] \to [s]$, $\sigma_2 : [s+t] \to [t]$. $\sigma_1(x)$ selects from some sequence x only those elements which have the form $in_1(u)$, while $\sigma_2(x)$ selects only the elements of the form $in_2(v)$. E.g. if $x = [in_1(a), in_2(b), in_2(c), in_2(d), in_1(e), in_2(f)]$, then $\sigma_1(x) = [a,e]$, $\sigma_2(x) = [b,c,d,f]$. σ_1 is defined by $\sigma_1(x) \stackrel{\text{def}}{=} flatten(\lambda(u).case\ u\ of\ in_1(u') \Rightarrow [u']\ in_2(u'') \Rightarrow [])(x)$, and σ_2 is defined similarly.

Operations on lists. *first* and *tail* can be defined by:

$$\begin{aligned} first(x) &\stackrel{\text{def}}{=} get(get(bm\_route(([()],[1,0]),\\ &\qquad split(x,[1, length(x) \dot{-} 1])))) \\ tail(x) &\stackrel{\text{def}}{=} get(bm\_route(([()],[0,1]),\\ &\qquad split(x,[1, length(x) \dot{-} 1]))) \end{aligned}$$

If x is empty, *split* will produce an error. Similary we can define *last* and *remove_last*, which return the last element, and delete the last element from a sequence, respectively. In general, we can access any element of some sequence of length n in $O(1)$ parallel time, and with $O(n)$ work complexity (we formally define below the time and work complexity). Using *map*, we can produce an arbitrary permutation in $O(1)$ parallel time, but with an increase of the work complexity to $O(n^2)$. Using radix sort in base n^ε, for some arbitrary $\varepsilon > 0$, we can even compute an arbitrary permutation in $O(1)$ parallel time with $O(n^{1+\varepsilon})$ work complexity. Alternatively, we can use an optimal sorting algorithm (see e.g. [Jaj92]), which reduces the work complexity to $O(n)$ by increasing the time complexity (e.g the sorting algorithm described in section 5 has $T = O(\log n \log\log n)$). Thus, the cost of performing an arbitrary permutation is visible in the higher level language.

The compilation theorem 7.1 is robust enough to hold if $\mathcal{NSC}$ is extended with additional primitives, like a general permutation *permute* or scan operations, provided that corresponding instructions are added to the BVRAM model. E.g theorem 7.1 can be extended to prove that $\mathcal{NSC}$ + *permute* can be efficiently compiled into BVRAM+*permute*. But in its present form theorem 7.1 is stronger, because it proves a general permutation is not necessary in a BVRAM in order to compile efficiently a high-level language like $\mathcal{NSC}$. This is of importance in view of the high cost of implementing a general permutation on existing massively parallel architectures [KLGLS90].

As promised, we will give a high-level definition of parallel time complexity T and work complexity W for $\mathcal{NSC}$ programs, in an machine independent way. The idea is for the parallel complexity of a program to be inferred from its structure in the same way in which the sequential complexity is inferred from the structure of a program in a sequential language. In our case, all primitive operations (including @ and *flatten*) take one parallel step, while in a $map(F)([x_0,\ldots,x_{n-1}])$, the n executions of F are done in parallel. The iteration construct however may count for several steps hence our definition cannot be done solely by induction on programs. This is handled by providing a formal operational semantics and then counting the depth of derivations in it. The work complexity is tied to the size of the data that is being manipulated.

Formally, we start by defining **S-objects** by the grammar: $C ::= () \mid n \mid (C,C) \mid in_1(C) \mid in_2(C) \mid [C,\ldots,C]$ where $n \in \mathbb{N}$. We only consider typed S-objects objects. We adopt a *unit size* complexity measure, and define **the size** of some S-object by $size(()) = size(n) = 1$, $size((C,D)) = 1 + size(C) + size(D)$, $size(in_1(C)) = size(in_2(C)) = 1 + size(C)$, $size([C_0,\ldots,C_{n-1}]) = 1 + \sum_{i=0,n-1} size(C_i)$. We use *true* and *false* as abbreviations for $in_1(())$ and $in_2(())$.

Next, we define the **evaluation** of some term (also called the operational semantics) in a natural semantics style, as in [Kah87]. This consists of rules which simultaneously define a binary relation $M \Downarrow C$ meaning that the term M evaluates to the S-object C and a ternary relation $F(C) \Downarrow C'$ meaning that the function F applied to the S-object C evaluates to C'. E.g. if $F = \lambda x.flatten(x)@[100]$ and $C = [[3,5],[2]]$, then $F(C) \Downarrow [3,5,2,100]$. Some representative rules are:

$$\frac{M \Downarrow m \quad N \Downarrow n}{M + N \Downarrow m + n}$$

$$\frac{M \Downarrow (C,D)}{\pi_1(M) \Downarrow C} \qquad \frac{M \Downarrow (C,D)}{\pi_2(M) \Downarrow D}$$

$$\frac{M \Downarrow C \quad N \Downarrow D}{(M,N) \Downarrow (C,D)} \qquad \frac{M \Downarrow C \quad F(C) \Downarrow D}{F(M) \Downarrow D}$$

$$\frac{M \Downarrow [C_0,\ldots,C_{m-1}] \quad N \Downarrow [D_0,\ldots,D_{n-1}]}{M@N \Downarrow [C_0,\ldots,C_{m-1},D_0,\ldots,D_{n-1}]}$$

$$\frac{F(C_0) \Downarrow D_0 \quad \ldots \quad F(C_{n-1}) \Downarrow D_{n-1}}{map(F)([C_0,\ldots,C_{n-1}]) \Downarrow [D_0,\ldots,D_{n-1}]}$$

$$\frac{P(C) \Downarrow false}{while(P,F)(C) \Downarrow C}$$

$$\frac{P(C) \Downarrow \mathit{true} \quad F(C) \Downarrow C' \quad \mathit{while}(P,F)(C') \Downarrow D}{\mathit{while}(P,F)(C) \Downarrow D}$$

The complete set of rules is given in [ST94] where we explain a technical complication caused by the presence of bound variables (lambda abstraction) in the language, namely the need to use *environments* as in [Cur88].

Thus, to *evaluate* some closed term M, one has to construct a *proof tree*, whose nodes are labeled with rules of the operational semantics, such that its root is labeled with some rule with conclusion $M \Downarrow C$. Based on this operational semantics, we now define the time and work complexity of $\mathcal{NSC}$ in a machine independent way.

Definition 3.1 *Consider some $\mathcal{NSC}$ term M. The* **time and work complexity** *$T(M), W(M)$ of $M \Downarrow C$ are defined by induction on the proof of $M \Downarrow C$. The induction is done simultaneously with the definition of the time and work complexity $T(F,C)$ and $W(F,C)$ of some evaluation $F(C) \Downarrow D$, where F is a $\mathcal{NSC}$ function, and C, D are S-objects. Except for the rules for map and while, for every rule of the form:*

$$\frac{M_1 \Downarrow C_1, \ldots, M_n \Downarrow C_n}{M \Downarrow C}$$

we define:

$$T(M) \stackrel{\text{def}}{=} 1 + \sum_{i=1}^{n} T(M_i) \qquad W(M) \stackrel{\text{def}}{=} \mathit{SIZE} + \sum_{i=1}^{n} W(M_i)$$

where SIZE is the total size of all S-objects mentioned in the rule (in the premises and the conclusion, including the environments). For the map-rule, the definition of W remains the same, while the definition of T becomes:

$$T(M) \quad \stackrel{\text{def}}{=} \quad 1 + \max_{i=1,n}(T(M_i))$$

(this corresponds to the fact that the function is applied in parallel *on all objects in the sequence). For the while rule we do not include in SIZE the size of the output D (otherwise, the final output D of while would be counted as many times as many iterations are performed by while). More precisely, if the last rule of $\mathit{while}(P,F)(C) \Downarrow D$ was:*

$$\frac{P(C) \Downarrow \mathit{true} \quad F(C) \Downarrow C' \quad \mathit{while}(P,F)(C') \Downarrow D}{\mathit{while}(P,F)(C) \Downarrow D}$$

then:

$$\begin{aligned} T(\mathit{while}(P,F),C) &\stackrel{\text{def}}{=} 1 + T(P,C) + T(F,C) + \\ &\quad\; T(\mathit{while}(P,F),C') \\ W(\mathit{while}(P,F),C) &\stackrel{\text{def}}{=} \mathit{size}(C) + \mathit{size}(C') + \\ &\quad\; W(P,C) + W(F,C) + \\ &\quad\; W(\mathit{while}(P,F),C') \end{aligned}$$

(i.e. $\mathit{size}(D)$ is not included explicitly in $W(\mathit{while}(P,F),C)$)

The language $\mathcal{NSC}$ together with its notions of time and work complexity is a model of parallel computation in its own right but parallel algorithms are most commonly given in terms of one of the several known flavors of PRAM. To facilitate comparisons, we offer the following efficient simulation ($\mathcal{NSC}$'s version of Brent's scheduling principle, as it were):

Proposition 3.2 *Any $\mathcal{NSC}$ function of time complexity T and work complexity W can be simulated on a CREW PRAM* with scan primitives *using p processors with asymptotic complexity $O(T + W/p)$.*

Proof. (Sketch) Given some function f in $\mathcal{NSC}$, first flatten f for an extended version of a BVRAM, with unbounded many vector registers and indirect addressing (essentially the VRAM of [Ble90], but with the communication primitives described in section 2). The resulting extended-BVRAM program has the same time and work complexity as f: see remark 7.3. Next use the simulation of an extended BVRAM on a CREW with scan primitives, in the spirit of [Ble90]. We need a CREW instead of a EREW in order to simulate *bm_route* and *sbm_route*. 2

4 Expressing *map*-recursive functions in $\mathcal{NSC}$

Although it is described in a concise, mathematical style (notice that we called it a "calculus" rather than a "language") $\mathcal{NSC}$ can be easily extended to a more user-friendly language, by allowing a certain amount of block structure: definitions of global/local variables and of nonrecursive functions. There is a straightforward translation of such an extension back into $\mathcal{NSC}$, which we omit from this extended abstract. Accomodating recursive functions though, is a more delicate problem, which we address here.

Consider the following limited form of recursion:

Definition 4.1 *A function definition is* ***map*-recursive** *if it has the form*

$$\mathsf{fun}\ f(x) = c(x, \mathit{map}(f)(d(x)))$$

First, it is easy for a compiler to check whether a recursive definition is of this form (in contrast, containment [Ble90] is an undecidable property). Second, this form is general enough to express many existent parallel algorithms: tail recursive definitions, and what is usually meant by divide-and-conquer recursion (for instance the worked example in section 5) are *map*-recursive. Here are some recursion schemata and a sketch of how to convert them into *map*-recursive form (and in the process "parallelize" them) :

$$\begin{array}{lcl} \mathsf{fun}\ g(x) & = & \mathsf{if}\ p(x)\ \mathsf{then}\ s(x)\ \mathsf{else}\ c(g(d_1(x)), g(d_2(x))) \\ \mathsf{fun}\ h(x) & = & \mathsf{if}\ p(x)\ \mathsf{then}\ s(x)\ \mathsf{else}\ c(h(d(x))) \\ \mathsf{fun}\ k(x) & = & \mathsf{if}\ p(x)\ \mathsf{then}\ s(x)\ \mathsf{else} \\ & & \mathsf{if}\ p'(x)\ \mathsf{then}\ c(k(d_1(x)), k(d_2(x))) \\ & & \mathsf{else}\ c'(k(d_1'(x)), k(d_2'(x)), k(d_3'(x))) \end{array}$$

For g, we construct a list of lenght 2, and recursively map g on it (Quicksort has this form). For h, the list will have length 1 (tail recursion is a particularization of this form). k is more interesting, since it divides its input into either two or three subproblems. Note that it is not contained [Ble90], so the compilation techniques described here work on some cases on which those of [Ble90] don't. In converting k, the list will have length 1, 3 or 4, where the first element is a tag, and k is slightly modified to return the identity on the tag (a sum of types is used here).

The first of our two main results states that *map*-recursion can be translated (in a source-to-source manner) into a $\mathcal{NSC}$ expression, while preserving its time complexity and "almost" preserving its work complexity.

Theorem 4.2 *Consider some function f defined in $\mathcal{NSC}$ extended with map-recursion, with time and step complexity T, W. Then, for any $\varepsilon > 0$, one can construct a function f' in $\mathcal{NSC}$ which is equivalent to f and which has time and work complexity $T' = O(T)$ and $W' = O(W^{1+\varepsilon})$ respectively. Moreover, if the divide and conquer tree of f is balanced, then $W' = O(W)$.*

Proof. For illustration, we consider only the function g from above. Suppose the types are: $g : s \to t$, $d_1, d_2 : s \to s$, and $c : t \times t \to t$. Not surprisingly, g can be expressed in $\mathcal{NRA}$, without recursion, in two steps, called *divide phase* and *combine phase* in [MH88]:

Divide Phase Start with the singleton sequence $y = [x]$ of type $[s]$, and apply repeatedly the function *flatten* $\circ$ *map*$(\lambda x.\mathsf{if}\ p(x)\ \mathsf{then}\ [x]\ \mathsf{else}\ [d_1(x), d_2(x)])$ having the type $[s] \to [s]$, until all its elements satisfy the predicate p. (We need to tag the elements resulting from $[x]$, to avoid applying p repeatedly on them; we omit the details.) Call y the resulting sequence.

Combine Phase Start by *map*-ing the function s on y, and then apply repeatedly c to adjacent elements of y: some additional bookkeeping is necessary to make sure c is applied to the correct pairs (e.g., it suffices to store the depth in the divide and conquer tree for each element in y, and only combine adjacent elements if they have the same depth). Stop when there is only one element in the resulting list.

Obviously, the translated g will have time complexity $O(T)$. The work complexity is also preserved, in the case in which the divide an conquer tree for the computation of $g(x)$ is perfectly balanced. When the is unbalanced, the leaves which are reached sooner have to coexists in the same sequence with those nodes which need more divide steps, thus adding to the total work complexity. Let ν be the number of different levels in the divide and conquer tree which contain leaves E.g. in an almost perfectly balanced tree, $\nu = 1$ or $\nu = 2$, while in a total "unbalanced" tree, ν can be equal to the total number of leaves, but still $\nu \leq W(g, x)$. We can compute ν in time and work complexity $O(T), O(W)$, by simulating only the *divide* phase, without retaining the results. Let $\varepsilon > 0$. We improve the *divide* phase, such that the time and work complexities of the translation of g into $\mathcal{NSC}$ become $O(T)$ and $O(\nu^\varepsilon W)$ respectively. Namely, we start with $\frac{1}{\varepsilon} + 1$ variables z_i, $i = 0, \ldots, \frac{1}{\varepsilon}$, initialized to $[]$, and with y initialized to the singleton $[x]$. We apply repeatedly the divide phase on y; whenever some leaves are reached, we move them into z_0. We only allow z_0 to be touched ν^ε times, after which we move its entire content into z_1, and empty z_0. We repeat this process, but only allow z_1 to be touched ν^ε times, at which point, we empty z_1, by moving everything into z_2. In general, we allow z_i to accumulate only ν^ε times, after which we empty it, by moving everything into z_{i+1}. Obviously, a number of $\nu^{i\varepsilon}$ levels of leaves must be discovered, before making one move into z_i; thus, $z_{\frac{1}{\varepsilon}}$ will be filled exactly once, with the leaves from all ν levels. To compute the total additional work complexity, observe that each leave travels exactly once through $z_0, z_1, \ldots, z_{\frac{1}{\varepsilon}}$, and in each z_i is "touched" exactly ν^ε times. Thus, the total work complexity is bounded by $(\frac{1}{\varepsilon} + 1)\nu^\varepsilon W = O(\nu^\varepsilon W)$. Of course, rather complicated bookkeeping is necessary to keep all elements in z_i sorted. The *combine phase* is done similarly, but in reverse. $\Box$

The technique of theorem 4.2 seems to extend to more general recursion schemas than the limited recursion. The main kind of recursion to which this technique does not apply is one in which some recursive call to f uses an argument which is computed with a recursive call itself, in the style of the Ackerman function: $A(x, y) = A(x-1, A(x, y-1))$. We argue that very few practical algorithms make indeed use of such recursion schemas.

5 An $O(\log n \log\log n)$ Mergesort Algorithm Expressed in $\mathcal{NSC}$

As evidence for the *practical* expressiveness of $\mathcal{NSC}$ we describe in it Valiant's fast mergesort algorithm [Val75, Jaj92], see the program in figures 1, 2, 3. As we have explained at the beginning of section 4 we are free to use block structure (we choose a syntax close to ML [MTH90]). More importantly, in view of theorem 4.2 we are free to use *map*-recursive definitions, or other recursive schemas which are convertible to *map*-recursion. The main function *mergesort* in figure 1 has the same recursion schema as the function g of section 4 and hence can be converted to a *map*-recursive form. Its parallel time complexity is $O(\log n \log\log n)$.

The fast, $O(\log\log m)$ time *merge* function exhibits a more complicated kind of *map*-recursion. To merge two sequences $A = [a_0, \ldots, a_{m-1}]$, $B = [b_0, \ldots, b_{n-1}]$, we divide A into $\lceil\sqrt{m}\rceil$ subsequences of length $\leq \sqrt{m}$; let $AA = [A_0, \ldots, A_{\sqrt{m}-1}]$ be the resulting nested sequence. Next, we find for each subsequence A_i the corresponding subsequence B_i in B, with which A_i has to be merged, and apply recursively *merge* on all pairs (A_i, B_i); let $BB = [B_0, \ldots, B_{\sqrt{m}-1}]$. Thus, the general structure of *merge* is:

```
fun merge(A, B) =
    if length(A) ≤ 2 then direct_merge(A, B)
    else let ... compute AA, BB as explained
         in flatten(map(merge)(zip(AA, BB))) end
```

which can be obviously translated into a *map*-recursion.

Figures 2 and 3 contain some auxiliary functions used in *merge*. The function $index(C, I)$ expects a sorted sequence of indexes $I = [i_0, \ldots, i_{k-1}]$ and, for $C = [C_0, \ldots, C_{n-1}]$, returns the sequence $[C_{i_0}, \ldots, C_{i_{k-1}}]$: it has constant time complexity and work complexity $= O(n + k)$. The function $index\_split(C, I)$ splits C according to the indexes in I, again provided that I is sorted, with similar time and work complexity. We use the construct $filter(P) : [t] \to [t]$, which for some predicate $P : t \to \mathbb{B}$ returns the sequence of all elements satifying P. It is expressibel in $\mathcal{NSC}$ by:

$$filter(P)(x) = flatten(map(\lambda u.\mathsf{if}\ P(u)\ \mathsf{then}[u]\ \mathsf{else}\ [])(x))$$

The functions *first*, *tail*, *last*, *remove_last* and *bm_route* are defined in section 3.

Using the techniques described in [Jaj92], the *merge* function can be transformed to become optimal, i.e. to reduce its work complexity from $O((m + n)\log\log m))$ to $O(m + n)$. This also gives us an optimal (i.e. with $O(n \log n)$ work complexity), $O(\log n \log\log n)$-time sorting function. The divide-and-conquer trees for both the sorting and the merging function are balanced, hence the translation of theorem 4.2 gives us an optimal $O(\log n \log\log n)$-time sorting function in $\mathcal{NSC}$.

```
fun mergesort(A) =
        if length(A) ≤ 1 then A
        else let val n = length(A)
                val AA = split(A, [n − n/2, n/2])
            in merge(mergesort(first(AA)),
                     mergesort(last(AA))))
            end

fun merge(A, B) =
    if length(A) ≤ 2 then direct_merge(A, B)
    else
    let val m = length(A)
        val n = length(B)
        val A' = sqrt_positions(A)
        val B' = sqrt_positions(B)
        (* A', B' have lengths √m and √n respectively *)
        val R' = direct_rank(A', B')
        val BB1 = sqrt_split(B)
                  (* split B into √n blocks *)
        val a_B = zip(A', index(BB1, R'))
                  (* group each a' with its block *)
        val RR' = map(rank_one)(a_B)
                  (* rank each a' in its block *)
        val R = map(λ(x, y).(x ∸ 1) * √n + y)
                  (zip(R', RR'))
        val AA = sqrt_split A
        val BB = index_split(B, R)
    in flatten(map(merge)(zip(AA, BB)))
    end
```

Figure 1: Valiant's $O(\log n \log\log n)$ sorting algorithm.

```
fun rank_one(a, B) = length(filter(λb.b ≤ a)(B))

fun direct_rank(A, B) = map(λa.rank_one(a, B))(A)

fun sqrt_positions(C) =
    let val n = length(C)
        val I = filter(λi.i mod√n = 0)(enumerate(C))
    in index(C, I)
    end

fun sqrt_split(C) =
    index_split(C, sqrt_positions(enumerate(C)))

fun direct_merge(A, B) =
    let val R = direct_rank(A, B)
        val BB = index_split(B, R)
    in first(BB)@
        flatten(map(λ(a, B).[a]@B)(zip(A, tail(BB))))
    end
```

Figure 2: Auxiliary functions used in *merge*.

```
fun index(C, I) =
    let val n = length(C)
        val k = length(I)
        val zero_to_k = enumerate(I)@[k]
        val delta_I = map(∸)(zip(I@[n], [0]@I))
        val P = bm_route((C, delta_I), zero_to_k)
        val delta_P = map(∸)(zip(P, remove_last([0]@P)))
    in bm_route((I, delta_P), C)
    end

fun index_split(C, I) =
    let val n = length(C)
    in split(C, map(∸)(zip(I@[n], [0]@I)))
    end
```

Figure 3: The functions *index* and *index_split*.

6 Theoretical Expressive Power

In this section we give evidence that $\mathcal{NSC}$ is not too restrictive, as a tool for designing parallel algorithms. Namely, let CRCW-TIME-PROC$(T(n), P(n))$ be the set of functions computable on a CRCW PRAM in time $T(n)$ using $P(n)$ processors, and $\mathcal{NSC}$-TIME-WORK$(T(n), W(n))$ the set of functions expressible in $\mathcal{NSC}$ with time and work complexity $T(n), W(n)$.

Proposition 6.1 *For $T(n), W(n)$, that are* suitable *(in the sense of [SV84]), we have:*

$$CRCW\text{-}TIME\text{-}PROC(O(T(n)), O(W(n))) \subseteq$$

$$\mathcal{NSC}\text{-}TIME\text{-}WORK(O(T(n)), W(n)^{O(1)})$$

More, we get equality, if in the definition of $\mathcal{NSC}$ we restrict the arithmetic operations to the set $\Sigma = \{+, \dot{-}\}$, and if we replace the unit size complexity *($size(n) \stackrel{\text{def}}{=} 1$ - see section 3) with the* logarithmic size complexity *($size(n) \stackrel{\text{def}}{=} \log n$), in the definition of the work complexity of $\mathcal{NSC}$.*

The proof uses a theorem in [SV84], credited to Ruzzo and Tompa, relating CRCW PRAM's to Alternating Turing Machines, and is omitted from this extended abstract. Using the above proposition and proposition 3.2 we can establish that NC coincides with the functions in $\mathcal{NSC}$ with polylogarithmic time and polynomial work complexity. Recall that $\mathcal{NSC}$ is parameterized by a set Σ of arithmetic operations.

Proposition 6.2 *Suppose all arithmetic operations in Σ are in NC. Then:*

$$NC = \mathcal{NSC}\text{-}TIME\text{-}WORK(\log^{O(1)} n, n^{O(1)})$$

7 Efficient Compilation of $\mathcal{NSC}$ to BVRAM

Theorem 7.1 (Compilation Theorem) *For every function f in $\mathcal{NSC}$ with time and work complexity T, W, there is a BVRAM, M, such that: $\forall \varepsilon > 0$, there is some program P for M, equivalent to f, having time complexity $T' = O(T)$ and $W' = O(W^{1+\varepsilon})$.*

Note that, in contrast to theorem 4.2, the number of registers only depends on f and not on ε. A *while*-construct can be rewritten as a tail recursive function, hence is *contained*, according to the definition in [Ble90], and therefore the compilation technique described there (for a VRAM, with unbounded many vector registers) preserves its step and work complexity. However, we cannot apply that compilation technique here. Indeed, when viewed as tail recursive function, the work complexity of *while* may increases significantly, because the final result after iterating n steps is touched n additional times, as the tail recursive function returns from its calls. In the definition of the work complexity for *while*, these n additional touches are not counted (see definition 3.1). So the tail recursive translation has a higher work complexity than the original *while* construct. We need a stronger compilation technique in order to stay within the lower work complexity. Moreover, we also only have a bounded number of vector registers.

The proof goes through the following steps:

- *Variable Elimination.* We translate $\mathcal{NSC}$ into a rather similar, but variable free language called Nested Relational Algebra, $\mathcal{NSA}$. The new language only contains *functions* $f s \to t$, i.e. no terms. Some term M in $\mathcal{NSC}$, of type t and with free variables $x_1 : s_1, \ldots, x_n : s_n$, will be translated into a function $f_M : s_1 \times \ldots \times s_n \to t$ in $\mathcal{NSA}$. The primitive functions and the constructs in $\mathcal{NSA}$correspong roughly to those in $\mathcal{NSC}$, with only one additional primitive: the function $\rho_2 : s \times [t] \to [s \times t]$ (see section 3 for its definition). The step and work complexity of functions expressed in $\mathcal{NSC}$ and $\mathcal{NSA}$ are the same. We omit the description of $\mathcal{NSA}$ from this extended abstract; it can be found in [ST94].

- *Flattening.* We define a language for flat sequences, called Sequence Algebra $\mathcal{SA}$, and translate $\mathcal{NSA}$ into $\mathcal{SA}$. Namely, for any $\varepsilon > 0$, we show how to translate a function f of $\mathcal{NSA}$ with time and work complexity T, W into an equivalent function in $\mathcal{SA}$ (thus using only flat types), with time and work complexity $O(T)$ and $O(W^{1+\varepsilon})$. Of course, any function in $\mathcal{SA}$ can be expressed in $\mathcal{NSA}$ with the same time and work complexity.

- We show that $\mathcal{SA}$ and BVRAM are equivalent, in the sense that any function in $\mathcal{SA}$ can be simulated by a BVRAM with the same time and work complexity, and conversely. One direction of this equivalence helps us completing the compilation, while the other direction allows us to perform optimizations at the level of the language $\mathcal{SA}$, instead of BVRAM.

7.1 The Sequence Algebra, $\mathcal{SA}$

The Sequence Algebra, $\mathcal{SA}$, only has *flat types.* More precisely, we define first *scalar types* by the grammar: $s ::= unit \mid \mathbb{N} \mid s \times s \mid s + s$, and next define the *flat types* by the grammar: $t ::= unit \mid [s] \mid t \times t \mid t + t$.

$\mathcal{SA}$ was designed by choosing some set of functions expressible in $\mathcal{NSA}$ (or, equivalent, $\mathcal{NSC}$) over flat types, which seemed to be enough to allow the language $\mathcal{NSA}$ to be translated (flattened) into $\mathcal{SA}$. In addition, $\mathcal{SA}$ is defined in an inductive way, which enables us to prove, by induction, properties about the functions expressible in $\mathcal{SA}$, e.g. lemma 7.2. $\mathcal{SA}$ stands in the same relationship to $\mathcal{NSA}$ as the relational algebra stands to the nested relational algebra [AB88].

Similar to $\mathcal{NSA}$, $\mathcal{SA}$ is a variable-free language, containing some primitive functions, and a set of rules for combining them in order to get more complex functions. We briefly describe $\mathcal{SA}$ below. A complete description of the language can be found in [ST94].

- Error, viewed as a function $\Omega : unit \to t$.

- *map*' s of scalar functions, $map(\varphi) : [s] \to [s']$, where $\varphi : s \to s'$ is a *scalar function*, i.e., informally, a function defined in $\mathcal{NSA}$ (or, equivalently $\mathcal{NSC}$) having only scalar types as input, output, and intermediate types, and without *while*.

- Operations on sequences: the empty sequence [], append @, length of a sequence, defined as $length(x) = [n]$, where n is the length of x, *zip*, *bm_route*, *sbm_route*, selections σ_1, σ_2 (see section 3), and the emptyness test *empty?*, of type $[s] \to \mathbb{B}$.

- Functions over flat types: the identity $id : t \to t$, composition of functions $g \circ f$, projections $\pi_i : t_1 \times t_2 \to t_i$, pairing of functions (f, g), injections $in_i : t_i \to t_1 + t_2$, and sum of functions $f_1 + f_2 : t_1 + t_2 \to t$, where $f_i : t_i \to t$ (an if construct can be derived from this). The latter is defined by: $(f_1 + f_2)(in_1(x)) \stackrel{\text{def}}{=} f_1(x)$ and $(f_1 + f_2)(in_2(x)) \stackrel{\text{def}}{=} f_2(x)$.

- Iteration: $while(p, f)$ is a function of type $t \to t$, whenever $f : t \to t$ and $p : t \to \mathbb{B}$ (recall that $\mathbb{B} = unit + unit$ and, thus, is a type of $\mathcal{SA}$).

As for $\mathcal{NSC}$ we define the the time and work complexity for some evaluation $f(C) \Downarrow$, where f is a function in $\mathcal{SA}$ and C is its input (a flat S-object). Note that in the absence of a general *map* there is no nested parallelism in $\mathcal{SA}$.

Although $\mathcal{SA}$ does not contain nested types, like $[\mathbb{N} \times [\mathbb{N} \times \mathbb{N}]]$, it is strong enough to allow such types to be encoded into flat types. The key technical tool for that is to encode some nonflat type $[t]$, where t is a flat type, by some flat type $SEQ(t)$. For this we use *segment descriptors*, as in [Ble90]. Formally, we transform some *flat type* t into another flat type $SEQ(t)$, defined by induction on t: (1) $SEQ(unit) \stackrel{\text{def}}{=} [\mathbb{N}]$ (2) $SEQ([s]) \stackrel{\text{def}}{=} [\mathbb{N}] \times [s]$, (3) $SEQ(t \times t') \stackrel{\text{def}}{=} SEQ(t) \times SEQ(t')$, (4) $SEQ(t + t') \stackrel{\text{def}}{=} [\mathbb{B}] \times SEQ(t) \times SEQ(t')$. The idea is that $SEQ(t)$, although a flat type, can encode sequences of elements from t, i.e. values of type $[t]$. The main technical fact enabeling us to prove efficient compilation is the following map lemma.

Lemma 7.2 (The Map Lemma) . *Let $f : t \to t'$ be some function in $\mathcal{SA}$, and let T, W be the time and work complexity of $map(f)$ (recall that $map(f)$ is in $\mathcal{NSC}$, but not in $\mathcal{SA}$). Then, for every $\varepsilon > 0$, there exists some function $SEQ(f) : SEQ(t) \to SEQ(t')$ in $\mathcal{SA}$, of time complexity $O(T)$ and work complexity $O(W^{1+\varepsilon})$, which simulates $map(f) : [t] \to [t']$. More, the structure of $SEQ(f)$ is independent of ε, which implies that "number of vector registers" used by $SEQ(f)$ is independent of ε.*

Proof. (Sketch) This is done by induction on the structure of f. When f is *map* of a scalar function, $SEQ(f)$ is essentially the same *map*. When f is some operation on a sequence, we only mention that $SEQ(empty?)$ is essentially a selection, $SEQ(\sigma_1)$ essentially σ_1, $SEQ(bm\_route)$ is a sbm_route, while $SEQ(sbm\_route)$ is another sbm_route. The only difficult case is when f is $while(p,g)$. We describe very informally how to compute $SEQ(while(p,g))(x)$, with $x = [x_0, \ldots, x_{n-1}]$, of a BVRAM. We could use the same idea as in theorem 4.2, but then the number of registers would depend on ε. Suppose x is in register V_0. We will use only two additional registers, V_1 and V_2, which are initially empty. Let t_i be the number of iterations of $while(p,g)(x_i)$, and assume without loss of generality that $t_0 < t_1 < \ldots < t_{n-1}$ (we conceptually group all x_i's having the same t_i), which implies $t_i \geq i$. Let $\delta = n^\varepsilon$, $w_i = W(while(p,g), x_i)$ and $r = \frac{1}{\varepsilon} - 1$. For the moment, assume that in the sequence $x_i, g(x_i), g^{(2)}(x_i), \ldots$, the last value (on position t_i) has the smallest size, denoted by s_i, so $s_i t_i \leq w_i$. The simulation proceeds in r stages. The first stage starts by repeatedly applying $SEQ(g)$ on x: whenever some x_i's reach the end of the iteration, move them into V_1, until the first $\frac{n}{\delta^r} (\leq n^\varepsilon)$ values are extracted from V_0, namely $x_i, i = 1, \frac{n}{\delta}$. The additional work complexity due to repeatedly touching the values in V_1 is $O(n^\varepsilon W)$. At this point, we move the entire V_1 into V_2. For each of the remaining stages $k = 1, r-1$, apply repeatedly $SEQ(g)$ on x, and move, when they terminate, the elements x_i, $i = \frac{n}{\delta^{r-k+1}}, \frac{n}{\delta^{r-k}}$ from V_0 to V_1: at the end of stage k, we move the entire V_1 into V_2. The additional work complexity due to repeatedly touching some element x_i in V_1 at this stage is $\leq s_i \frac{n}{\delta^{r-k}}$. But since $i \geq \frac{n}{\delta^{r-k+1}}$, we have that $t_i \geq i \geq \frac{n}{\delta^{r-k}} \frac{1}{\delta}$, hence the additional work complexity for x_i is $\leq s_i t_i \delta \leq w_i n^\varepsilon$, which, when added up, accounts for only $O(n^\varepsilon W)$ for stage k, which adds up to at most $O(\frac{1}{\varepsilon} n^\varepsilon W) = O(n^\varepsilon W)$ for all r stages. During all r stages, V_2 is touched only r times, for an additional $O(W)$ work complexity. At the end of the last stage, all x_i's ($i = 1, n$) end up in V_2, so V_2 contains the result of $SEQ(while(p,g))(x)$.

Finally we have to show how to define $SEQ(while(p,g))(x)$ in the general case, when the sequence $x_i, g(x_i), g^{(2)}(x_i), \ldots, g^{(t_i)}$ has a minimum size on some position m_i which is not necessarily the last one. In that case we first compute m_i, for each i: this can be done with complexities $O(T)$ and $O(W)$, by simply applying $SEQ(g)$ repeatedly, and eliminating those elements which reach the end of their iteration. Next we split the whole iteration $SEQ(while(p,g))(x)$ in two parts, essentially by synchronizing the n parallel iterations at the moment when they reach their minimum size, namely: (1) perform the n parallel iterations, as described above, but stop the iteration over x_i at stept m_i, (2) continue the n parallel iterations, from step m_i to t_i, using the same technique, but in reverse (because now the minumum sizes are at the beginning).

2

Remark 7.3 *Had we had arbitrarily many registers instead of a bounded number, we could have designed $SEQ(f)$ with time and work complexity $O(T)$ and $O(W)$ (instead of $O(T)$ and $O(W^{1+\varepsilon})$, which is used in the proof of proposition 3.2. Indeed, for $f = while(p,g)$, assume again that, $\forall i = 1, n$, the smallest size, denoted s_i, in the sequence $x_i, g(x_i), g^{(2)}(x_i), \ldots, g^{(t_i)}(x_i)$ is on the last position. Then $SEQ(while(p,g))$ is simulated by placing, upon completion, each element x_i in some different register V_i. At the end we have to combine the registers $V_1, \ldots, V_n$, which we do in the following order: combine V_n with V_{n-1}, the result with V_{n-2}, ..., the result with V_1. The additional work complexity for the combine phase due to x_i is $s_i i$, which bounded by w_i, because of our assumption about s_i. We can extend the simulation to the case when the smallest sizes s_i are reached at arbitrary moments, using the same technique as above.*

Finally, we flatten the language $\mathcal{NSA}$ into $\mathcal{SA}$. We start by flattening the types. For every type s of $\mathcal{NSA}$, we define $COMPILE(s)$ to be a flat type, which encodes s. Namely:

$$\begin{aligned}
COMPILE(unit) &\stackrel{\text{def}}{=} unit \\
COMPILE(\mathbb{N}) &\stackrel{\text{def}}{=} [\mathbb{N}] \\
COMPILE(s \times s') &\stackrel{\text{def}}{=} COMPILE(s) \times COMPILE(s') \\
COMPILE(s + s') &\stackrel{\text{def}}{=} COMPILE(s) + COMPILE(s') \\
COMPILE([s]) &\stackrel{\text{def}}{=} SEQ(COMPILE(s))
\end{aligned}$$

Also, we define the functions $encode_s : s \to COMPILE(s)$ and $decode_s : COMPILE(s) \to s$ in $\mathcal{NSA}$, with time complexity $O(1)$ and work complexity linear in the size of the input, with the property $decode_s(encode_s(x)) = x$, for every $x \in s$. The definition of the functions *encode* and *decode* are rather standard, and are omitted from this extended abstract.

Finally, we can prove:

Proposition 7.4 *Let $f : s \to s'$ be some function in $\mathcal{NSA}$ with time and work complexity T, W. Then, for every $\varepsilon > 0$, there is some function $COMPILE(f) : COMPILE(s) \to COMPILE(s')$ in $\mathcal{SA}$ which "simulates f", i.e. for every x, $COMPILE(f)(encode(x)) = encode(f(x))$, with time and work complexity $O(T)$, $O(W^{1+\varepsilon})$. Moreover, f' requires "the same number of BVRAM registers" for every ε.*

Proof. (Sketch) By induction on the structure of f. All cases are straightforward, except for the case when $f = map(g)$, where we use the Map lemma. 2

7.2 Equivalence of $\mathcal{SA}$ and BVRAM

The types in $\mathcal{SA}$ are slightly richer than those of the BVRAM: $\mathcal{SA}$ allows for types like $[unit + \mathbb{N} + \mathbb{N} \times \mathbb{N}] + [\mathbb{N} \times \mathbb{N}] \times [\mathbb{N}] + unit$, while the types on the BVRAM are only of the form $[\mathbb{N}] \times \ldots \times [\mathbb{N}]$. However, encoding of $\mathcal{SA}$ types into BVRAM types is straightforward.

Proposition 7.5 *$\mathcal{SA}$ and BVAM are equivalent, i.e. any function f in $\mathcal{SA}$ with time and work complexity T, W can be simulated on a BVRAM with the same time and work complexity, and conversely.*

Proof. Simulating some function of $\mathcal{SA}$ by a BVRAM program is easily done by induction on the structure of that function. The converse is slightly more involved. Indeed, let r be the number of registers of a BVRAM M, and h some function in $\mathcal{SA}$ of type $[\mathbb{N}] \times ([\mathbb{N}])^r \to [\mathbb{N}] \times ([\mathbb{N}])^r$ performing *one step* of the program of M (where the program counter is encoded by a singleton sequence, on the first position). By iterating h we indeed achieve the desired time complexity, but not the work complexity, since at each step, the function h touches all r registers. To avoid this, we define a sequence of r functions f_i, $i = 1, r$. The inputs and outputs for f_i are: the values of the i "smallest" resgisters, at some particular moment, the indexes of these i registers, the size S of the next largest register, and the program counter. f_i iterates the one-step function as long as it only affects the i registers it sees, and as long as all the i sizes stay less than S. If any of these conditions is violated, f_i stops. To do its job, f_i calls f_{i-1}, which iterates steps on M by only looking at the smallest $i-1$ registers: when f_{i-1} finishes, f_i tries to do one more step by taking into account the i's smalles register as well, which f_{i-1} ignores. If it cannot, then it returns (to f_{i+1}). Else, it performs the operation, and calls f_{i-1} again, possibly with a different set of $i-1$ registers, from the set of i registers it sees. $\Box$

Although only one direction of proposition is actually needed for the compilation theorem 7.1, the converse is significant from the point of view of optimizations: it implies that any optimizations done for the BVRAM can also be performed at the level of the $\mathcal{SA}$ language.

8 Conclusions

We intend to use $\mathcal{NSC}$ as a core for a "real" parallel language for querying nested collections, by adding proven features such as those encountered in functional languages like ML. Guaranteed complexity bounds such as those emerging from this paper can serve as useful guidelines for language design, especially in the database area. Of course, the techniques we have used in the translation of *map*-recursion and in the unnesting of nested parallelism need to be validated by practical implementations. Equally important is to continue to investigate the practical expressiveness of $\mathcal{NSC}$ by attempting to represent various known efficient parallel algorithms. Another direction of investigation is to develop optimization techniques for this language by using ideas that have been proved useful in databases.

References

[AB88] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. In *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988. Also available as INRIA Technical Report 846.

[BBW92] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, Massachusetts, 1990.

[Ble93] Guy Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

[BS90] Guy Blelloch and Gary Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[BTS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[Cur88] P. L. Curien. The $\lambda\rho$-calculus: An abstract framework for environment machines. Technical Report URA 725, Laboratoire d'Informatique, Departement de Mathematiques et d'Informatique, Ecole Normale Superieure, 45 Rue d'Ulm, 75230 Paris Cedex 05, France, 1988.

[HS86] Daniel Hillis and Guy Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[Jaj92] Joseph Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[KLGLS90] Kathleen Knobe, Joan D. Lukas, and Jr. Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures : Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992.

[MH88] Zhijing Mou and Paul Hudak. An algebraic model for divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2:257–278, 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[SBT94] Dan Suciu and Val Breazu-Tannen. A query language for NC. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, Minneapolis, Minnesota, May 1994. To appear. See also UPenn Technical Report MS-CIS-94-05.

[ST94] Dan Suciu and Val Tannen. Efficient compilation of high-level data parallel algorithms. Technical Report MS-CIS-94-17, University of Pennsylvania, Philadelphia, PA 19104, April 1994. Available by ftp from `ftp.cis.upenn.edu`.

[SV84] Larry Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal of Computing*, 13:409–422, May 1984.

[Val75] L. G. Valiant. Parallelism in comparison problems. *SIAM Journal of Computing*, 4(3):348–355, 1975.