December 1994

A Shared-Memory Multiprocessor Implementation of Data-Parallel Operators for ML

Dan Suciu
University of Pennsylvania

Lorenz Huelsbergen
University of Pennsylvania

University of Pennsylvania Institute for Research in Cognitive Science Technical Report No. IRCS-94-27.

A Shared-Memory Multiprocessor Implementation of Data-Parallel Operators for ML

Abstract

We have designed and implemented an asynchronous *data-parallel* scheduler for the SML/NJ ML compiler. Using this general scheduler we built a data-parallel module that provides new operators to manipulate *sequences* (i.e., arrays, vectors) in parallel. Parallelization concerns such as thread creation and synchronization are hidden from the application programmer by ML's module abstraction. We find that languages with modules, higher-order functions and automatic parallel storage management can, in this manner, seamlessly support data-parallel operators. An implementation of applications using the new sequence module on an eight-processor shared-memory machine indicates that in some cases useful speedup is possible with our approach.

Comments

University of Pennsylvania Institute for Research in Cognitive Science Technical Report No. IRCS-94-27.

The Institute For Research In Cognitive Science

A Shared-Memory Multiprocessor Implementation of Data-Parallel Operators for ML

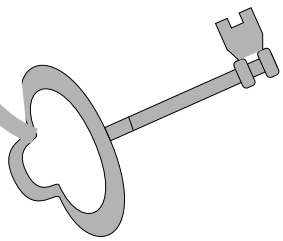
by

Dan Suciu
University of Pennsylvania
Lorenz Huelsbergen
AT&T Bell Laboratories

University of Pennsylvania
3401 Walnut Street, Suite 400C
Philadelphia, PA 19104-6228

December 1994

Site of the NSF Science and Technology Center for Research in Cognitive Science



University of Pennsylvania
Founded by Benjamin Franklin in 1740

IRCS Report 94-27

A Shared-Memory Multiprocessor Implementation of Data-Parallel Operators for ML

Dan Suciu
University of Pennsylvania
suciu@saul.cis.upenn.edu

Lorenz Huelsbergen
AT&T Bell Laboratories
lorenz@research.att.com

Abstract

We have designed and implemented an asynchronous *data-parallel scheduler* for the SML/NJ ML compiler. Using this general scheduler we built a data-parallel module that provides new operators to manipulate *sequences* (*i.e.*, arrays, vectors) in parallel. Parallelization concerns such as thread creation and synchronization are hidden from the application programmer by ML's module abstraction. We find that languages with modules, higher-order functions and automatic parallel storage management can, in this manner, seamlessly support data-parallel operators. An implementation of applications using the new sequence module on an eight-processor shared-memory machine indicates that in some cases useful speedup is possible with our approach.

1 Introduction

General purpose parallel processors in the form of small-scale shared-memory computers are rapidly becoming commonplace, not only as compute servers and workstations, but also as personal computers. Parallel software techniques and parallel languages, however, primarily target parallel supercomputers. This is because the users of such large-scale parallel machines seek the best performance possible. Hence, they are often willing to adopt novel, even exotic, programming languages. In contrast, programmers of relatively small machines desire programming languages that primarily provide portability across many general architectures and through many generations of a specific architecture. Performance—especially parallel performance—is currently a less pressing concern when programming commodity machines.

The data-parallel model [HS86] of computation extends existing languages (*e.g.*, Fortran [KLS+91] and C [Thi93]) with new constructs that operate on aggregate data in parallel. However, with new language constructs come new semantics (*cf.* HPF and C*). We favor the introduction of parallel programming on machines with a small number of processors (*e.g.*, $P = 2,4,8$) through the addition of *data-parallel modules* to existing general-purpose languages. The advantage of this approach is that the language's familiar syntax and semantics remain unchanged. A data-parallel module[1] contains common abstractions for manipulating aggregate data structures; for example, a data-parallel module for arrays would contain functions to operate on array elements in parallel. Since abstraction mechanisms (*cf.* [Mac84]) can hide the details of a module's (parallel) implementation, the application programmer need not be concerned with parallelization issues such as synchronization, load balance and parallel thread creation. In this data-parallel model, the programmer

[1] A collection of data-parallel modules encompassing a multitude of aggregate data types constitutes a *data-parallel library*.

writes sequential code as before, but can now transparently tap the underlying machine's parallel processors to improve performance by using the operators supplied in data-parallel modules.

We have designed a *data-parallel scheduler* for the ML language. ML is a modern higher-order call-by-value language with powerful type and module systems along with automatic memory management via garbage collection. Data-parallel modules can in turn be implemented on top of this data-parallel scheduler for various data types. We have implemented the scheduler in the Standard ML of New Jersey compiler (SML/NJ) [AM87, App92] with multiprocessor extensions [MT93]. The multiprocessor extensions provide mechanisms for thread creation and synchronization, and support parallel memory allocation. The ML language meets our basic requirements: it provides powerful data and procedural abstraction mechanisms, it's programs are extremely portable, and its programming model abstracts from underlying memory systems. The SML/NJ compiler also meets our basic requirements: it is practical (it is being used to implement systems software [BHL+94]), and it generates optimized native code for many commodity processors (*e.g.*, X86, PowerPC, Mips, Sparc, Alpha).

Using our data-parallel scheduler we have implemented a data-parallel module that provides a data structure called a *sequence* and common functions that manipulate sequences. A sequence is an ordered collection of homogeneous elements, much like an array or a list, and differs from the latter only by the operations associated with it, and their cost. The data-parallel `map` function is the main function of the module; it takes as arguments a function and a sequence, applies the function in parallel to each element of the sequence, and returns a new sequence of results. The module also includes variants of `map`, as well as first-order parallel functions, like `append` or `flatten`. All module functions are deterministic. To ensure deterministic behavior, restrictions apply to the *higher-order* data-parallel functions; *i.e.*, to the functions in the module that accept arbitrary functions as arguments. *E.g.* `(map f)` is valid only when `f` has no side-effects; this restriction ensures that the result does not depend on the order in which `f` is applied. While we do not currently enforce determinism in our system, this can be done automatically using effect inference [TJ92, LG88].

We also implemented a sequential version of the module having the same signature (*i.e.*, interface) as the parallel one. The sequential implementation is faster than the parallel implementation when running on a single (`P`$= 1$) processor. It is useful for running programs on a sequential machine.

Using the sequence module, we have programmed several applications. We report the results of data-parallel execution for a matrix package containing addition, multiplication and inverse; quicksort; odd-even mergesort; a set package; and an algorithm for computing the Voronoi diagram of a set of points. Matrix inversion and sorting show only slight speedup using the data-parallel sequence module. Matrix multiplication, the set operations, and the Voronoi-diagram computation exhibit useful speedup.

Section 2 briefly presents the ML language, the multiprocessor extensions to the SML/NJ compiler, and gives an example of an algorithm expressed with our data-parallel functions. Section 3 describes the interface to the data-parallel module. Section 4 describes the data-parallel scheduler. Measurements of applications using the data-parallel sequence module are reported in section 5. Section 6 describes an alternative sequence implementation.e discuss related work in section 7 and future work in section 8.

2 Preliminaries

In this section we briefly describe the Standard ML language, the multiprocessor ML compiler we are using, the sequence data type, and the underlying analytic complexity model. As an example of ML and of data-parallel sequences, we then give a data-parallel quicksort function for sequences.

2.1 Standard ML

Standard ML (SML) [MTH90, Mil78] is a *higher-order call-by-value* language with automatic storage management via *garbage collection*. Though "mostly" functional, ML permits imperative assignment of mutable value cells called references. SML has *static polymorphic type inference*. Type inference can, in most cases, deduce the types of a program's expressions without programmer annotation; polymorphic types enable code reuse since, for example, a single polymorphic function can be used to map functions over different lists with different element types. Additionally, SML has a powerful typed *module system* that admits safe separate recompilation [AM94]. Since ML types and modules are used heavily in the following exposition, we describe their pertinent features here.

In addition to the base types `int`, `real`, `bool`, `string`, and `unit`[2], ML permits user-defined data types (*e.g.*, a `list` data type), tuples, records, and function types. To illustrate, the type of a function from binary tuples of integers to a boolean is written: `(int * int)->bool`. Polymorphic types, written `'a`,`'b`,..., are also possible. For example, the polymorphic type `'a list` describes a list carrying elements of some (yet unspecified) type `'a`.

An SML module (see, *e.g.*, [Mac84]) consists of a `structure` containing data-type declarations and function definitions. Additionally, a module may also be a `functor`; a `functor` is simply a structure parameterized by other structures. Associated with every structure S is a `signature` that gives the *visible* types of the definitions in S. A signature specifies a module's interface. We can therefore completely specify the interfaces of our data-parallel modules by presenting only their signatures.

2.2 Multiprocessor SML/NJ

Our data-parallel modules are implemented using the Standard ML of New Jersey (SML/NJ) [AM87, App92] optimizing ML compiler. The compiler extends SML with mutable arrays, vectors (immutable arrays) and first-class continuations. Extension to arrays and vectors are vital to our sequence implementation; we use continuations [Wan80] to cleanly introduce parallel threads into an otherwise sequential compiler.

Our data-parallel scheduler is derived from the **Procs & Locks** system [MT93] for SML/NJ which provides mechanisms for thread creation (using SML/NJ's continuations) and thread synchronization. **Procs & Locks** permits concurrent allocation of data into separate heaps. However, garbage collection is currently done sequentially in this system. All processors synchronize before a garbage collection; at this point, a single processor proceeds and performs the entire collection.

Reported measurements are of SML/NJ with **Procs & Locks** running on a shared-memory (bus based) SGI Challenge with eight 150Mhz R4400 processors.

2.3 Sequences

Sequences are defined to be ordered collections of homogeneous elements. The elements are 0-based, thus the notation[3] for a sequence of length n is $<x_0, \ldots, x_{n-1}>$. We denote the empty sequence as $<>$. The main source of parallelism offered by our data-parallel sequence module is in processing the n elements of a sequence in parallel; via, for example, parallel `map` and related functions. Nested sequences are allowed *e.g.*, the sequence:

$$<<a, b>, <c, d, e>, <f>, <>, <g, h>>$$

[2] The type `unit` is the type containing a single value, written `()`.

[3] For clarity, we will use this sequence notation in ML expressions although ML's syntax does not support it.

Combining parallel `map` with nested sequences leads to *nested parallelism*, which is a powerful technique for expressing parallel algorithms [Ble90].

2.4 Parallel Complexity Measures

In order to guide the design of efficient parallel algorithms, we adopt the *parallel-time, total-work* complexity model [Ble90, Jaj92]. This model associates two complexity measures with a program: the *parallel-time complexity* T, and the *total-work complexity* W. The first measure intuitively corresponds to the running time of a program executed on an ideal parallel machine, with arbitrarily many processors, capable of fully exploiting all the parallelism of a given algorithm. *E.g.* for the expression `(map f` $<x_0, \ldots x_{n-1}>$`)`, T is defined to be:

$$T(\texttt{(map f } <x_0, \ldots x_{n-1}>\texttt{)}) \stackrel{\text{def}}{=} 1 + \max_{i=0,n-1}(T(\texttt{(f } x_i\texttt{)}))$$

The total work complexity W of a program is defined to be the total number of operations performed by that program. In the case of the `(map f` $<x_0, \ldots x_{n-1}>$`)` construct, W is defined to be:

$$W(\texttt{(map f } <x_0, \ldots x_{n-1}>\texttt{)}) \stackrel{\text{def}}{=} n + \sum_{i=0,n-1}(W(\texttt{(f } x_i\texttt{)}))$$

These two complexities T and W for a given program are used to estimate the running time T_P of that program on a multiprocessor with P processors. A good implementation will thrive to achieve:

$$T_P = O(T + \frac{W}{P})$$

However, achieving good load balance may come at a cost not included in the above formula.

In our setting, the number of processors is low: typically $P = 2$ or $P = 4$. The dominant term in T_P is then $\frac{W}{P}$. In order to be efficient, parallel algorithms need to be designed with low work complexities W. However our experiments (see Section 5) show that the T component has a major importance as well. The T component roughly corresponds to the number of synchronization steps, which are much more expensive than computation steps. We measured speedups for algorithms with $T = O(1)$ or $T = O(\log n)$, but we could not achieve speedup for algorithms with $T = O(\log^2 n)$ or $T = O(n)$.

2.5 Example: Quicksort

Nested data-parallelism [Ble93] yields an elegant implementation of the divide-and-conquer parallelism. The example of figure 2.5, adapted from [Ble93], is a divide-and-conquer parallel algorithm for `quicksort`.

The above quicksort function is written in ML syntax extended with sequences. The functions `append`, `filter`, `length`, `map`, `sequenceOfList`, and `sub` operate on sequences and are from the data-parallel `SEQUENCE` module described in §3. The `filter` function's first argument is a higher-order function that is notated as `(fn x => ...)` in ML. The parameter to `quicksort` has been explicitly declared to be an integer sequence. An application of quicksort binds the local identifier `q` to a nested *sequence of integer sequences*. The expression

```
(map quicksort (sequenceOfList [s1, s3]))
```

applies `quicksort`—a parallel function itself—in parallel to the subsequences `s1` and `s3` respectively.

```
fun quicksort (s:int seq) =
  let val n = length s
  in if n <= 1 then s
  else
    let val pivot = sub(s,n div 2)
        val s1 = filter (fn x => x < pivot) s
        val s2 = filter (fn x => x = pivot) s
        val s3 = filter (fn x => x > pivot) s
        val q = map quicksort (sequenceOfList [s1, s3])
  in append(append(sub(q,0),s2),sub(q,1))
  end
```

Figure 1: Data-parallel quicksort

3 The Data-Parallel Sequence Module

The data-parallel module consists of an ML structure called `SEQUENCE`. The signature for `SEQUENCE` is given in Figure 2. The structure defines the sequence type constructor `seq`, the number of active processors `P`, an exception `SEQUENCE` to be raised in case of an error, and a collection of data-parallel functions. We first describe the collection of data-parallel functions. The functions are grouped in the following categories:

Parallel-map functions: `map`, `mapi`, `sequence`, `filter`, `filteri`. These function apply a higher-order function in parallel to all elements of a sequence. They are defined as follows:

$$\begin{aligned}
\texttt{(map f } <x_0, \ldots, x_{n-1}>\texttt{)} &\stackrel{\text{def}}{=} <\texttt{f(}x_0\texttt{), ..., f(}x_{n-1}\texttt{)}> \\
\texttt{(mapi f } <x_0, \ldots, x_{n-1}>\texttt{)} &\stackrel{\text{def}}{=} <\texttt{f(}x_0\texttt{,0), ..., f(}x_{n-1}\texttt{,}n-1\texttt{)}> \\
\texttt{(sequence f n)} &\stackrel{\text{def}}{=} <\texttt{f 0, f 1, ..., f(n-1)}>
\end{aligned}$$

Two `filter` functions return the subsequence of values satisfying a given predicate:

$$\begin{aligned}
\texttt{(filter p } <x_0, \ldots, x_{n-1}>\texttt{)} &\stackrel{\text{def}}{=} <x_i \mid \texttt{p(}x_i\texttt{) = true}> \\
\texttt{(filteri p } <x_0, \ldots, x_{n-1}>\texttt{)} &\stackrel{\text{def}}{=} <x_i \mid \texttt{p(}x_i\texttt{,}i\texttt{) = true}>
\end{aligned}$$

Restructuring functions: `append`, `flatten`, `zip`, `subSeq`, `evenSeq`, `oddSeq`, `shuffle`, `split`.
The functions `append` and `flatten` are defined as:

$$\begin{aligned}
\texttt{append}(<x_0, \ldots, x_{m-1}>, <y_0, \ldots, y_{n-1}>) &\stackrel{\text{def}}{=} <x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1}> \\
\texttt{flatten}(<x_0, \ldots, x_{m-1}>) &\stackrel{\text{def}}{=} \texttt{append}(x_0, \texttt{append}(x_1, \ldots, \texttt{append}(x_{n-1}, <>)\ldots))
\end{aligned}$$

The function `zip` is defined as:

$$\texttt{zip}(<x_0, x_1, \ldots, x_{n-1}>, <y_0, \ldots, y_{n-1}>) \stackrel{\text{def}}{=} <(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})>$$

```
signature SEQUENCE =
  sig
     type 'a seq
     exception SEQUENCE
     val P : int

  (* Parallel maps *)
     val map :  ('a -> 'b) -> ('a seq -> 'b seq)
     val mapi :  ('a * int -> 'b) -> ('a seq -> 'b seq)
     val sequence :  (int -> 'a) -> (int -> 'a seq)
     val filter :  ('a -> bool) -> 'a seq -> 'a seq
     val filteri :  ('a * int -> bool) -> 'a seq -> 'a seq

  (* Restructuring functions *)
     val append :  'a seq * 'a seq -> 'a seq
     val flatten :  'a seq seq -> 'a seq
     val zip :  'a seq * 'b seq -> ('a * 'b)seq
     val subSeq :  'a seq -> (int * int) -> 'a seq
     val evenSeq :  'a seq -> 'a seq
     val oddSeq :  'a seq -> 'a seq
     val shuffle :  'a seq * 'a seq -> 'a seq
     val split :  'a seq * int seq -> 'a seq seq

  (* Utilities *)
     val emptySeq :  'a seq
     val isEmpty :  'a seq -> bool
     val length :  'a seq -> int
     val sub :  'a seq * int -> 'a
     val sequenceOfList :  'a list -> 'a seq

  (* Reduce and scan *)
     val reduce :  'b * ('a -> 'b) * ('b * 'b -> 'b) -> ('a seq -> 'b)
     val scan :  'b * ('a -> 'b) * ('b * 'b -> 'b) -> ('a seq -> 'b seq)
  end
```

Figure 2: The signature of the data-parallel sequence module

The expression `zip(x,y)` raises the exception `SEQUENCE` when `x` and `y` are of different lengths. The functions `subSeq, evenSeq, oddSeq`, and `shuffle` are defined as:

$$\begin{aligned}
\texttt{subSeq } <x_0, x_1, \ldots> \texttt{ (i,n)} &\stackrel{\text{def}}{=} <x_i, x_{i+1}, \ldots, x_{i+n-1}> \\
\texttt{evenSeq}(<x_0, x_1, x_2, \ldots>) &\stackrel{\text{def}}{=} <x_0, x_2, x_4, \ldots> \\
\texttt{oddSeq}(<x_0, x_1, x_2, \ldots>) &\stackrel{\text{def}}{=} <x_1, x_3, x_5, \ldots> \\
\texttt{shuffle}(<x_0, x_1, \ldots>, <y_0, y_1, \ldots>) &\stackrel{\text{def}}{=} <x_0, y_0, x_1, y_1, \ldots>
\end{aligned}$$

The function `subSeq` raises the exception `SEQUENCE` when the indices `i,n` are "out of range"; `shuffle` raises the exception `SEQUENCE` when $\texttt{(length(x)} \neq \texttt{length(y))} \wedge \texttt{(length(x)} \neq \texttt{length(y) + 1)}$. Finally, the function `split` is defined as:

$$\begin{aligned}
\texttt{split}(<x_0, \ldots, x_{n-1}>, <i_0, \ldots, i_{m-1}>) &\stackrel{\text{def}}{=} \\
&<<x_0, \ldots, x_{i_0-1}>, <x_{i_0}, \ldots, x_{i_1-1}>, \ldots, <x_{i_{m-2}}, \ldots, x_{i_{m-1}-1}>>
\end{aligned}$$

For example, when $\texttt{s} = <a, b, c, d, e>$ and $\texttt{x} = <2, 0, 3>$, then $\texttt{split(s,x)} = <<a, b>, <>, <c, d, e>>$. The function raises exception `SEQUENCE` when $m \neq \sum_{k=0,m-1} i_k$. The restructuring functions are parallel, but their degree of parallelism is inherently limited. For small sequences it is likely that the necessary synchronization overheads will offset any benefit gained from parallelism.[4]

Utility functions: `emptySeq`, `isEmpty`, `length`, `sub`, `sequenceOfList`. The utility functions are sequential functions that offer access to the sequence abstraction. They are defined by:

$$\begin{aligned}
\texttt{emptySeq} &\stackrel{\text{def}}{=} <> \\
\texttt{isEmpty(s)} &\stackrel{\text{def}}{=} \textit{true} \text{ iff } \texttt{s} = <> \\
\texttt{length}(<x_0, \ldots, x_{n-1}>) &\stackrel{\text{def}}{=} n \\
\texttt{sub}(<x_0, \ldots, x_{n-1}>\texttt{,i)} &\stackrel{\text{def}}{=} x_i \\
\texttt{sequenceOfList}([x_0, \ldots, x_{n-1}]) &\stackrel{\text{def}}{=} <x_0, \ldots, x_{n-1}>
\end{aligned}$$

While ML has no notation for sequences, it does have one for lists, namely $[x_0, \ldots, x_{n-1}]$. Thus, `sequenceOfList` provides a mechanism for explicit sequence notation in ML; instead of `<1,2,3>`, we write:

```
sequenceOfList [1,2,3]
```

Reduce and scan functions: The function calls `reduce(e,f,g)` and `scan(e,f,g)` expect the higher-order function `g` to be commutative and associative. Then, denoting $\texttt{g}(u, v) = u * v$, `reduce` and `scan` are defined by:

$$\begin{aligned}
\texttt{reduce (e,f,g)} <x_0, \ldots, x_{n-1}> &\stackrel{\text{def}}{=} \texttt{f}(x_0) * \ldots * \texttt{f}(x_{n-1}) \\
\texttt{scan (e,f,g)} <x_0, \ldots, x_{n-1}> &\stackrel{\text{def}}{=} <e, f(x_0), f(x_0) * f(x_1), \ldots, f(x_0) * \ldots * f(x_{n-2})>
\end{aligned}$$

The commutativity and associativity conditions imposed on `g` are undecidable and hence cannot be checked by a compiler [BTS91]. The commutativity condition on `g` is motivated by our goal of an efficient implementation rather than by the semantics of `reduce` and `scan`.

[4] We experimented with sequential restructuring functions and observed only negligible changes in execution times.

The sequence module is itself an ML functor:

```
functor sequence(val P : int) = ...
```

In order to initialize the data-parallel module, one invokes `sequence` with the desired number of processors. Thus:

```
structure sequence = sequence(val P=4)
```

will activate three processors in addition to the one currently active.

3.1 Implementation of Sequences

An early version of our implementation represented a sequence as an ML array. The `P` processors could access and update different regions of the array independently and in parallel. This simple implementation did not perform well, due to two factors. First, SML/NJ records each update of an array element in a store list[5] update operations costly. The second reason is concerns cache coherence. Suppose the array elements `a[i]` and `a[j]`, `i < j`, are processed by different processors. If `j-i` is smaller than the cache line of the SGI (16 bytes), then this line may be present in both processors cache. Updating either `a[i]` or `a[j]` will trigger the cache coherency maintenance protocol, reducing the overall performance. We found the delays caused by these issues to be significant in the case of short sequences; hence we turned to SML/NJ's vectors.

SML/NJ offers an immutable version of arrays called *vectors*. Their content is fixed at creation time, and cannot be updated in parallel.

We used a two-level implementation of sequences, which combines the efficiency of vectors, with the parallel access capability of arrays. Recall that `P` is the number of active processors. We implement a sequence $s = <x_0, \ldots, x_{n-1}>$ as an array of `P` vectors. Each vector will contain a subset of the elements of s, in a round-robin fashion. *E.g.* when `P` $= 2$, s will be represented by an array of length two containing the vectors $<x_0, x_2, x_4, \ldots>$ and $<x_1, x_3, x_5, \ldots>$.

Thus, the definition of the `seq` type constructor inside the module abstraction is:

```
'a seq = 'a Vector.vector Array.array
```

Access to the element with index i is straightforward:

```
fun sub(s,i) = Vector.sub(Array.sub(s,i mod P), i div P)
```

Here `Array.sub(s, i mod P)` returns the element `i mod P` of the array `s`. To improve performance, we impose `P` to be a power of two, and replace `mod` and `div` with bit-wise operations.

This representation is well suited for parallel processing of sequences with `P` processors: each processor `i` will perform computations on all sequence elements stored in vector `i`. However, we observed a performance improvement in some of our benchmarks if we partition a sequence into only two vectors, even when `P` > 2. The explanation for this behavior is that most of these benchmarks are divide-and-conquer parallel functions. The input sequence x to such a function is divided into two subsequences, x_0 and x_1, to which the function is applied recursively and in parallel. The main source of parallelism consists therefore in applying the parallel `map` to the nested sequence $<x_0, x_1>$. By partitioning this sequence into `P` vectors, `P` > 2, we end up doing unnecessary work for the empty `P` $- 2$ vectors. In addition, near the leaves of the divide-and-conquer tree, the

[5] Generational garbage collector require store lists containing all updated cells in order to live data due to pointers from old generations to younger ones.

recursive function is applied to short sequences x (`length(`x`)` ≤ 2) and there is the overhead for splitting it into `P` vectors when `P` > 2. The measurements reported in Section 5 are based on partitioning of sequences into two vectors.

On the other hand, by partitioning sequences into only two vectors we run the danger of not being able to fully exploit a machine with `P` > 2 processors. This may happen in expressions with only one level of parallelism ($T = 1$), *e.g.* in `(map f s)`, with `f` a non-parallel function. We argue that for most problems there will be enough work for each processor of a small multiprocessor. Indeed, even the simplest parallel algorithms have at least $T = 2$ or $T = 3$ levels of parallelism—enough work for `P` $= 4$ or `P` $= 8$ processors. *E.g.* in the case of matrix addition, where matrixes are represented as sequences of sequences of reals, we have $T = 2$ and we can keep `P` $= 4$ processors busy. For matrix multiplication, $T = 3$. Other parallel algorithms typically have $T = \log\log n$, $T = \log n$, *etc.*

3.2 Parallel-Map Implementations

To construct in parallel a sequence s of length n we proceed as follows. First we allocate an array `v` of length `P` and fill it with empty vectors. Next we activate the `P` processors (servers). Each processor is assigned to $\frac{n}{\mathtt{P}}$ elements of s. Namely, processor j sequentially constructs the vector $<s[j], s[j+P], s[j+2P], \ldots>$, and assigns the result to `v[`j`]`. The functions `map, mapi`, and `sequence` are implemented in similar fashion.

The motivation behind the two level implementation of sequences is the following. In ML arrays are mutable data structures. However each update is recored in a store list, thus degrading the overall performance (see Section 5). By contrast vectors are immutable, and, hence, more efficient. However, for the same reason, vectors cannot be constructed in parallel. The only way of constructing a vector is by using the build-in function `Vector.tabulate`, which is sequential. The two level implementation of sequences takes advantage of the parallelism enabled by the mutable arrays while preserving the efficiency of vectors.

3.3 Restructuring Implementations

The round-robin distribution of the elements of a sequence among its components enables us to design a simple and efficient implementation for some of the restructuring functions, namely `append, flatten, zip, subSeq, split, evenSeq, oddSeq,` and `shuffle`.

For example, to compute `s'' = append(s,s')` we append each of the `P` vectors of `s` with precisely one of the `P` vectors of `s'`. Namely we append `Array.sub(s,j)` with `Array.sub(s',(j+n) mod P)`, where `n` is the length of `s`. This algorithms clearly defines `P` independent tasks for the `P` processors.

Computing `flatten,` require some more work. Namely let `s' = flatten(s)`, i.e. `s'` is an array of `P` vectors. We observe that for any subsequence $\mathtt{s_i}$ of `s` and for any `j = 0, P-1`, all elements of the vector $\mathtt{s_i}$`[j]` will end up in the same vector, say `s'[k]` of the result `s'`, where `k` depends not only on `i` and `j`, but also on the length of the sequences $\mathtt{s_0}, \ldots, \mathtt{s_{i-1}}$. So each processor `k`, for `k=0,P-1`, will sequentially construct a vector with all elements of those vectors $\mathtt{s_i}$`[j]` which have to end up in `s'[k]`. Finally `filter, filteri` are implemented using `flatten`.

The round-robin distribution could cause more cache misses when the elements of a sequence larger than the cache are processed sequentially, because consecutive elements in the sequence lie in different cache lines.

```
signature SCHEDULER =
   sig
      val P : int
      val do_n :  (int -> unit) -> (int -> unit)
   end
```

Figure 3: The signature for the data-parallel scheduler

3.4 `REDUCE` and `SCAN` Implementations

The functions `reduce` and `scan` also admit simple implementation provided that the binary operation `g` is commutative. We compute `reduce(e,f,g)(s)` as follows. First we create an array `v` of length `P`. Next we activate the `P` processors. Processor i applies the function `g` on the ith vector, and stores the result in `v`[i]. Finally we combine the elements of `v` sequentially.

4 The Data-Parallel Scheduler

At the core of a data-parallel library is a distributed data-parallel scheduler, written in ML using SML/NJ's (non-standard) arrays, vectors, continuations, multiprocessor extensions, and bit-wise operations. Data-parallel modules and data-parallel libraries constructed from such modules can share a common scheduler. This section describes the details of the scheduler design.

During initialization, the scheduler acquires `P-1` processors (in addition to the active one), and starts a *server* on each of them. An application program can be now in one of two states: (1) a *data-parallel* state, when it is executing data-parallel functions, and (2) a *program* state, when it is not executing data-parallel functions. During the the *program* state, all `P-1` servers are dormant, while the main program thread runs on one processor. Any call to a data-parallel function generates additional threads, and awakens the `P-1` dormant servers. All `P` processors start to execute a part of the available work; the program is now in the *data-parallel* state. New data-parallel functions may be called while in this state, *e.g.* in the case of nested parallelism, and this creates more work for the servers. Eventually all work will be finished, and the server that completes the last piece of work resumes the program's main thread of execution.

The scheduler has the signature given in Figure 3. `P` is the number of processors. The expression `(do_n f n)` will compute `f 0, f 1, ..., f(n-1)` in parallel, by distributing the `n` independent threads to the `P` processors. Recall that nested parallelism is possible, *i.e.* that each of the `n` initial threads may create other threads, again by calling other instances of `do_n`.

The goal of the scheduler is to distribute as uniformly as possible the active threads among the `P` servers, while performing few synchronizations and with little overhead.

The scheduler maintains a unique pool of all available work. This pool is further described below. When a sever becomes idle, it extracts a *chunk* of work from this pool. If the pool is empty, then the server enters the dormant state. The scheduler uses two locks (supplied by the OS via **Procs & Locks**). The first lock serializes accesses to the pool; each server must acquire this lock before entering the critical region and release it afterwards. A second lock keeps the servers dormant while the pool is empty.

A pool entry contains information about an active `(do_n f n)` call. More precisely, it contains the following:

- `f`, the function to be called.
- `unallocated`, the number of threads not yet allocated to servers. Initially `unallocated = n`; a server decrements this count when it extracts work from this pool entry before starting the new work. The entry is removed from the pool when `unallocated` becomes zero.
- `k`, the continuation of the `(do_n f n)` expression.
- `unfinished`, the number of threads still to be completed. Initially `unfinished = unallocated = n`. A server decrements `unfinished` after completing its current work. Afterwards, if `unfinished` goes to zero, the server continues with `k`. Otherwise, it tries to get a new piece of work from the pool. The following invariant is preserved: $0 \leq \texttt{unallocated} \leq \texttt{unfinished} \leq \texttt{n}$.
- `chunk`, a constant equal to $\frac{\texttt{n}}{\texttt{P}}$. In order to reduce the number of synchronizations, a server will extract `chunk` number threads from a pool entry, rather than only one. Thus, the server will set
$$\texttt{unallocated} := \texttt{unallocated} - \texttt{chunk}$$
before starting its piece of work, and
$$\texttt{unfinished} := \texttt{unfinished} - \texttt{chunk}$$
after it finishes.
- `ex`, a reference to an exception option. Its initial value is `NONE`. Whenever an exception `e` is raised by one of the `n` threads, this field is set to `(SOME e)`. The last server checks this field before continuing with `k`, and, if it is set, raises the exception `e` in the continuation[6] `k`.

The function `do_n` first creates a new entry in the pool. It then transforms itself into a server. It is possible that a different server (on a different processor) will service the last chunk of work from this entry and continue with its continuation. Thus the program thread may migrate to another processor.

An access to the pool is costly because a server needs to be in a critical region during that time. We use two techniques to reduce the number of accesses to the pool:

1. As described above, each server executes a number of $\frac{\texttt{n}}{\texttt{P}}$ function calls before making another access to the pool. We chose this number because when the program only executes a single `(do_n f n)` then the `n` threads are evenly divided among the `P` servers, and each server needs to access the pool only once. The disadvantage of this technique is that it may lead to a load imbalance when the `n` function calls compute for unequal amounts of time.
2. Before creating a new entry in the pool, the function `do_n` checks whether there are any dormant servers to service the new entry. If not, it then executes the first thread `f(n-1)` by itself, and checks again for dormant servers to service `(do_n f (n-1))`. It may end up performing all `n` threads sequentially, in the case where the other servers are busy, thus avoiding unnecessary synchronizations; the test for dormant servers needs no synchronization. In consequence, the pool will rarely have more than one entry. We experimented with other strategies for avoiding overloading, *e.g.* checking whether the size of the pool is less than some constant, but they did not perform as well in practice.

The scheduler module is a functor:

```
functor Scheduler(val P : int) :  SCHEDULER
```

Upon instantiation of the `Scheduler` functor, `P - 1` new processors are acquired and fresh servers initiated.

[6] A useful feature, which unfortunately is missing in SML/NJ. We had to simulate it.

Description	$P = 1$	$P = 2$	$P = 4$
Matrix multiplication	44.54	27.14	18.95
Matrix inversion	9.04	9.06	9.38
Quicksort	1.03	0.95	0.87
Odd-even mergesort	1.86	2.09	1.55
Set operations	9.33	6.62	4.90
Voronoi diagram	0.60	0.44	0.33

Figure 4: Benchmark timings in seconds

5 Benchmarks

5.1 Measurement techniques

For each benchmark we took three measurements: (1) using a sequential version of the library, (2) using the data-parallel library with two processors, and (3) using the data-parallel library with four processors. All times are wall-clock times and so include the time spent in the sequential garbage collector.

On one processor, the sequential library is much faster than the parallel library running with only a single processor. Hence, the speedups reported here are "true" speedups—speedups relative to a pure sequential implementation.

In the current implementation of the multiprocessor extension of SML/NJ, the garbage collection is sequential. Each processor has its own *allocation arena* (set to 1Mb) where it can freely allocate without synchronization. When a processor exhausts its arena, it waits until the other processors also exhaust their respective arenas A single processor then runs the garbage collector. The overhead introduced by this skew is serious because it increases the sequential portion of the program. We are investigating alternate techniques (*e.g.*, software polling to detect when a processor's arena requires collection) to mitigate this skew caused by unequal allocation rates. We furthermore suspect that a system with a parallel garbage collector could reduce the running times of our benchmarks significantly.

Figure 4 contains the running times, in seconds, for the six benchmarks. Matrix multiplication, set operations, and Voronoi diagram exhibit useful speedup. Their speedup is plotted in Figures 5-7. On each graph we also plotted the expected speedup, derived from the analytic complexity model $T_P = T + \frac{W}{P}$. The values for T and W were computed using the least square method. All benchmarks were run with version 106 of the SML/NJ compiler, except for the Voronoi benchmark which was run under version 104.

5.2 Basic Routines

5.2.1 Matrix Package

We implement a matrix as a sequence of *rows*, where each row is a sequence of real numbers. The direct implementation of matrix multiplication has parallelism at three levels: (1) all rows of the output matrix are computed in parallel, (2) all columns of a row are computed in parallel, and (3) the inner product uses the parallel function `reduce`. A description of this algorithm is given in figure 5.2.1. Its parallel-time complexity for multiplying two $n \times n$ matrixes is $T = O(1)$ (assuming that `reduce` takes $T = O(1)$), and its parallel work complexity is $W = O(n^3)$. The benchmark was run on a 250×250 matrix.

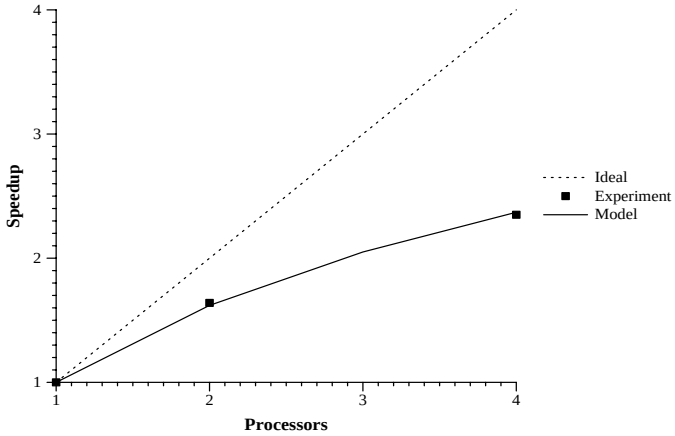


Figure 5: Matrix Multiplication

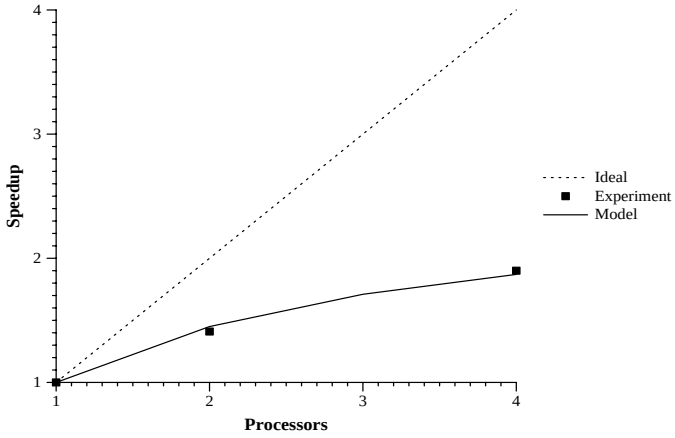
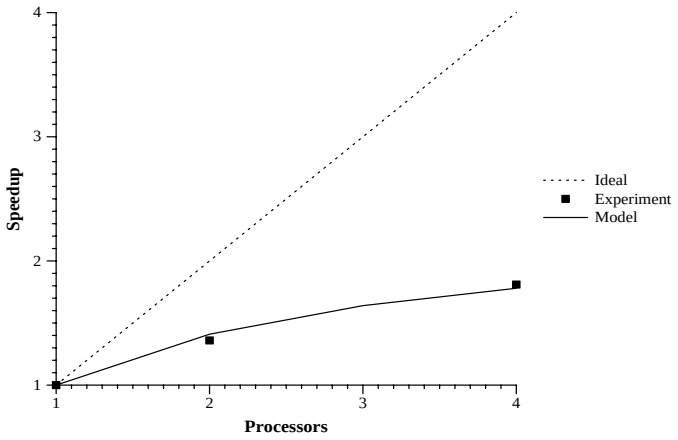


Figure 6: Set Operations



Voronoi

Figure 7: Voronoi

```
type 'a matrix = 'a seq seq

fun matrixMultiply(m1,m2) =
    let val m2' = transpose m2
        val inner = (reduce (0.0, op *, op +)) o zip
        fun innerWith s = map (fn s' => inner(s,s'))
    in map (fn s => innerWith s m2') m1
    end
```

Figure 8: Matrix multiplication: $T = O(1), W = O(n^3)$. Here `g o f` is the ML notation for the composition of the functions `f` and `g`; thus `inner` is a function. Also, `op +` is the ML infix notation for the function `f(x,y) = x + y`. The function `transpose` transposes a matrix; its definition is not shown.

Matrix inversion uses Gauss' elimination method. It takes $T = O(n)$ parallel steps and a total of $W = O(n^3)$ work complexity to invert a matrix of size $n \times n$. Since sequences are immutable data-structures, we need to allocate $O(n)$ new matrixes. This benchmark was run using a 100×100 matrix.

5.2.2 Sorting

We implemented two data-parallel sorting algorithms: quicksort and odd-even merge-sort.

The quicksort algorithm is described in section 2.5. It takes on average $T = O(\log n)$ parallel steps for a total work complexity $W = O(n \log n)$. Odd-even merge-sort has parallel time complexity $T = O(\log^2 n)$ and parallel work complexity $W = O(n \log^2 n)$. Both are divide-and-conquer algorithms, but odd-even merge-sort has two levels of divide-and-conquer: the first for sorting and the second for merging the two sorted halves. Thus the execution tree of the odd-even merge-sort has $O(\log^2 n)$ leaves, compared to only $O(\log n)$ leaves for quicksort. Note that the leaves correspond to light computation threads, while the nodes closer to the root correspond to heavier threads. We sort $n = 1000$ 10-tuples of real numbers, with a comparison function involving floating-point operations.

We obtained speedup with quicksort. Odd-even mergesort shows less speedup because the large number of small threads require too many synchronizations which offset the speedup gained by the coarser-grained parallel threads. Compilation techniques like those described in [HLA94] could improve the performance of odd-even merge-sort by avoiding the parallelization of light threads.

5.2.3 Set Operations

We implemented a module for operations on homogeneous sets. A set is represented as a sorted sequence. The user supplies the order relation for the base types, *e.g.* $\leq$ for integers. For sets of more complex types, like sets of sets of integers, the order relation is lifted from the base types to set types or to product types. The signature is for this set module is given in figure 9. Note how this module builds on the basic sequence module.

The function `order(lt,eq)` returns an abstract representation of an order relation, while `liftOrder(ordr)` lifts an order relation from some type `t` to `t set`. Once a set is created, *e.g.* via `setOfSequence`, the new set retains the order relation with which it was created. The functions `flatten` and `product` automatically generate the order on the resulting set. The binary operations make sense only if the two sets have the same order relation—this condition is enforced at run time. We compare structurally two order relations. At base types we use the following trick. The function `order` associates a reference to `unit` to each order relation on base types, which uniquely identifies that particular order relation. We compare two order relations at base types by comparing this reference cell.

Binary operations such as `union` and `difference` that use odd-even merge-sort have $O(\log n)$ parallel-time and $O(n \log n)$ total-work complexity.

Our benchmark evaluates

```
union(x, map (fn y0 => difference (x0, y0)) y)
```

where `x` and `y` are sets containing 100 randomly generated sets of integers with cardinalities ranging between zero and 20, and where `x0 = flatten x`. The reported times are the average of runs using four different random input data.

```
signature SET =
  sig
     exception Set

     structure sequence :  SEQUENCE

     type 'a set
     type 'a order

     val order :  ('a * 'a -> bool) * ('a * 'a -> bool) -> 'a order
     val liftOrder :  'a order -> 'a set order
     val empty :  'a set
     val isEmpty :  'a set -> bool
     val setOfSequence :  'a order * 'a sequence.seq -> 'a set
     val sequenceOfSet :  'a set -> 'a sequence.seq
     val set :  'a order * (int -> 'a) * int -> 'a set
     val member :  'a * ('a set) -> bool
     val included :  ('a set) * ('a set) -> bool
     val equal :  ('a set) * ('a set) -> bool
     val union :  ('a set) * ('a set) -> ('a set)
     val difference :  ('a set) * ('a set) -> ('a set)
     val intersect :  ('a set) * ('a set) -> ('a set)
     val flatten :  'a set set -> 'a set
     val map :  ('a -> 'a) -> ('a set -> 'a set)
     val product :  ('a set) * ('b set) -> (('a * 'b)set)
     val filter :  ('a -> bool) -> ('a set -> 'a set)
     val exists :  ('a -> bool) -> ('a set -> bool)
     val forall :  ('a -> bool) -> ('a set -> bool)
  end
```

Figure 9: The signature for the set module

5.3 Voronoi-Diagram Computation

We compute the Voronoi diagram of `n=650` points in a plane using essentially the algorithm of [GS85]. It is a divide-and-conquer algorithm, with parallel time complexity $T = O(n)$ and work complexity $W = O(n \log n)$. It assumes that the points are sorted lexicographically; sorting the points is not part of the benchmark.

6 An Alternative Sequence Implementation

In this section we describe an alternate method of implementing data-parallel sequences using *flat parallelism*.

6.1 Flat parallelism

Flat parallelism is a program's top-level parallelism; *i.e.*, it is the parallelism generated when the program is in the *program state* (§4). All other parallelism, *i.e.* that encountered when in a *data-parallel* state, will, under flat-parallelism, evaluate sequentially. *E.g.*, in computing the expression `(map f <`$\mathtt{x}_0$`, ..., `$\mathtt{x}_{n-1}$`>)`, n parallel threads are created; during the computation of each thread however, no additional threads are created even though data-parallel functions may be called in these computations—flat parallelism restricts parallelization to the top level. The flat-parallelism evaluation strategy is efficient when top-level parallelism generates more parallel threads than the number of available processors.

The main advantage of a flat-parallelism sequence implementation is that it requires only a simple—and hence potentially efficient—data-parallel scheduler. The `P-1` servers wait at a barrier for a client to generate some work. When `n` independent parallel threads are generated (at the top level), each of the `P` servers (recall that the client becomes on of the servers) executes `n/P` tasks, and checks into a second barrier. After the second barrier completes, the `P-1` servers enter a dormant state waiting for the next piece of work. All the while, the client is executing the main program thread.

For matrix operations, this implementation of the `sequence` library performs better than the nested-sequence implementation (§3) because, as is to be expected, much parallelism is available at the top level in such operations.

6.2 Flat sequences - an implementation in C

The flat-parallelism approach is also attractive because, as [Ble90] showed, it can directly address the problem of *load balancing*. The idea is to restrict all sequences to flat sequences, and to restrict the function `f` in higher-order constructs, such as `(map f)`, `(sequence f)`, *etc.*, to primitive operations (*e.g.*, $+, -, *, /$, *etc.*). Only flat parallelism will then arise and the simple scheduler of §6.1 is guaranteed to achieve perfect work balance, since all threads will require the same number of computation steps.

We designed an additional data-parallel SML/NJ module for flat sequences, called `flatsequence`. The signature of `flatsequence` is different from that of `sequence`; notably, the parallel `map` function is no longer present, and this new module only admits sequences of integers or of reals. The signature is given in Figure 6.2.

The types `seq` and `rseq` denote sequences of integers and reals respectively. The expression `init(P)` acquires `P-1` processors (in addition to the existing one), and `close()` releases the previously allocated processors. The expression `sequence(n,v)` creates a constant sequence of length `n` containing `v`, while `tabulate(n,f)` sequentially creates the sequence `<f 0, f 1, ..., f(n-1)>`.

```
signature FLATSEQUENCE =
    sig
        type seq
        type rseq
        val init :  int -> unit
        val close :  unit -> unit

    (* Operations on sequences of integers *)
        val sequence :  int * int -> seq
        val tabulate :  int * (int -> int) -> seq
        val sequenceOfList :  int list -> seq
        val add :  seq * seq * seq -> unit
        val mult :  seq * seq * seq -> unit
        val sub :  seq * int -> int
        val length :  seq -> int
        val append :  seq * seq * seq -> unit
        val bmRoute :  seq * seq * seq -> unit
        val sbmRoute :  seq * seq * seq * seq -> unit
        val reducePlus :  seq -> int
        val sreducePlus :  seq * seq * seq -> unit
        val permute :  seq * seq * seq -> unit

    (* Operations on sequences of reals *)
        val rsequence :  int * real -> rseq
        val rtabulate :  int * (int -> real) -> rseq
        val rsequenceOfList :  real list -> seq
        val radd :  rseq * rseq * rseq -> unit
        val rmult :  rseq * rseq * rseq -> unit
        val rsub :  rseq * int -> real
        val rlength :  rseq -> int
        val rappend :  rseq * rseq * rseq -> unit
        val rbmRoute :  rseq * rseq * seq -> unit
        val rsbmRoute :  rseq * seq * rseq * seq -> unit
        val rreducePlus :  rseq -> real
        val rsreducePlus :  rseq * rseq * seq -> unit
        val rpermute :  rseq * seq * rseq -> unit
    end
```

Figure 10: The signature of a data-parallel module for flat sequences

Parallelism stems from the data-parallel functions `add`, `mult`, `append`, `bmRoute`, `sbmRoute`, `reducePlus`, and `sreducePlus`, as well as from their real-number counterparts. Thus `add(a,b,c)` assigns `a` $\leftarrow$ `b + c`, where `b+c` means component-wise addition of the sequences `b` and `c`; `mult` performs data-parallel multiplication in a similar fashion. The expression `append(a,b,c)` assigns to `a` the sequence obtained by appending `b` and `c`. The expression `bmRoute(a,b,c)` stores in `a` the elements of `c` replicated a number of times dictated by the integers in `b`. *E.g.* if `b=<2,0,3>` and `c=<x,y,z>`, then after `bmRoute(a,b,c)`, we have `a = <x,x,z,z,z>`. Note that the lengths of `b` and `c` must coincide, while the length of `a` must be equal to the sum of elements in `b`. The expression `sbmRoute(a,b,c,d)` is similar, but works on *segmented* sequences. Here the pair `c,d` should be viewed as a nested sequence of integers, where `d` is the sequence of lengths, while `c` is the flattened sequence. The effect of `sbmRoute` is to replicate the subsequences of `c` a number of times dictated by `b`. *E.g.* when `b=<2,0,3>` and `c=<3,2,2>, d=<x,x',x'',y,y',z,z'>`, then, after `sbmRoute(a,b,c,d)`, `a` will hold `<x,x',x'',x,x',x'',z,z',z,z',z,z'>`. The function `reducePlus` computes the sum of the elements in a sequence, wile `sreduce` is its segmented version. Namely `sreducePlus(a,b,c)` considers the pair `b,c` as a nested sequence and stores in `a` the sums of elements of each of the subsequences of `b,c`. *E.g.* when `b=<x,x',z,z',z''>, c=<2,0,3>`, then the sequence `<x+x', 0, z+z'+z''>` will be stored in `a`. Finally, `permute(a,b,c)` assigns `a[b[i]]` $\leftarrow$ `c[i]`, for all `i`.

Each data-parallel function on sequences of integers has a corresponding function for sequences of reals.

The functions in the `flatsequence` module were written in C, but they are intended to be called only from ML programs. In particular, our implementation incorporates the new flat data-parallel functions directly into the SML/NJ runtime system. The multiprocessor support for SML/NJ was not needed for this implementation since the C functions can call the operating system's library functions for multiprocessor management directly.

For the flat-parallelism approach to data-parallel sequences, operations on nested sequences, or the nested parallelism arising from recursive function calls, must be flattened, using (*e.g.*) the techniques of [Ble90], before they can be operated upon by the flat sequence library. Flattening is a complicated process, but in theory it could be done such as to guarantee perfect load balance. It is not supported in our current implementation.

We tested a matrix multiplication algorithm with $T = O(n)$ and $W = O(n^3)$ on matrixes of size 250×250. The algorithm is essentially the hand-flattened version of the function `matrixMultiply` of figure 5.2.1, with one change: the main parallel `map` expression (`map (fn s => innerWith s m2') m1`) is replaced with a sequential loop of n steps (assuming n to be the number of lines in `m1`). Thus the parallel time complexity becomes $T = O(n)$, instead of $T(1)$ for `matrixMultiply`, where each parallel step has a work complexity of $O(n^2)$. We made this change in order to reduce the memory requirement for each parallel step from $O(n^3)$ to $O(n^2)$. The absolute running times (and the consequent speed-up) for up to 6 processors were better than those of our nested-parallelism data-parallel sequence implementation. The timings for this flat-parallelism program are in Figure 6.2, and the speedups are given in Figure 11.

We tried to reduce the additional memory requirement even further, from $O(n^2)$ to $O(n)$, by sequentializing the parallel `map` at the next level (in the `innerWith` function); the resulting $T = O(n^2)$ algorithm did not perform as well, due to the total $O(n^2)$ synchronizations required.

7 Related Work

Data-parallel programming is advocated in [HS86], where a collection of data-parallel algorithms can be found. Our data-parallel primitives and the complexity measures T, W are in the spirit of NESL described in [Ble93, BC93]. NESL is a parallel functional language with sequences as central types. It has only some limited form of higher-order constructs, and is therefore not a higher-order language in the sense of ML.

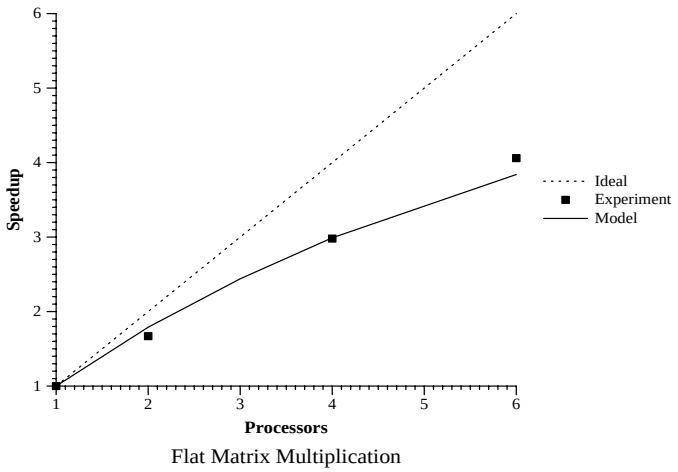


Figure 11: Matrix multiplication using flat sequences

N	Sequential	P = 2	P = 4	P = 6
250	11.87	7.12	3.98	2.92

Figure 12: Benchmarks timings for matrix multiplication using the flat-sequence module

Polymorphism is handled by specializing a polymorphic function for specific types, and generating code for each type. NESL's implementation mainly targets massively parallel architectures or vector processors, such as CM2, CM5, Cray C90, and it follows a different approach from ours, based on *flattening nested parallelism*.

The data-parallel core P of Proteus [MNP+91, GMN+94, PP93] handles nested sequences and nested parallelism. Its implementation follows the technique of NESL, with a more algebraic flattening technique.

SISAL [FC90, Ske91, Feo91] is a general-purpose applicative language, featuring nested sequences with `map` parallelism, streams with lazy evaluation, record and union types. It is only first order. A highly-optimized compiler on shared-memory architectures was developed for Sisal [FC90], with performance comparable to that of FORTRAN. Our implementation follows some of the ideas of SISAL, but is much simpler due to two factors: (1) SML/NJ is a stack-free implementation (all data is heap allocated), hence we need not distinguish between threads with a stack and threads without, (2) SML/NJ does not have explicit storage deallocation, hence there is no need for SISAL's *storage deallocation list*.

Data-parallel ML [FVH94, Hai93, HF93] is an extension of ML for a SIMD model of computation, intended as an intermediate target for more elaborate data-parallel languages and machines. Process creation and nested parallelism are not supported in Data-parallel ML.

C* [Thi93] and High Performance Fortran [KLS+91] are extensions of the languages C and FORTRAN respectively, featuring low-level data-parallel operations on flat sequences, which are called *parallel variables* in C*, and *arrays* in HPF.

8 Conclusion and Future Work

Our work shows that a relative simple implementation of data-parallel functions added to a powerful language like SML/NJ can offer a simple model of parallel computation with useful speedups on a small-scale, shared-memory commodity multiprocessors. The sequential garbage collection of SML/NJ is still however a serious bottleneck for in our implementation. We plan to integrate our data-parallel functions in future versions of SML/NJ that support concurrent or parallel garbage collection.

References

[AM87] A. W. Appel and D. B. MacQueen. A standard ml compiler. *Functional Programming Languages and Computer Architecture*, 1987.

[AM94] A. Appel and D. B. MacQueen. Separate compilation for standard ml. In *ACM Sigplan Conference on Programming Language Design and Implementation*, June 1994.

[App92] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[BC93] G. E. Blelloch and S. Chatterjee. Implementation of a portable nested data-parallel language. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–112, San Diego, May 1993.

[BHL+94] E. Biagioni, R. Harper, P. Lee, and B. G. Milnes. Signatures for a protocol stack: A systems application of standard ML. In *ACM Conference on Lisp and Functional Programming*, June 1994.

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

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

[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.

[FC90] J. T. Feo and D. Cann. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, pages 349–365, 1990.

[Feo91] J. T. Feo. Arrays in Sisal. In Lenore Mullin, Michael Jenkins, Gaetan Hains, Robert Bernecky, and Guang Goa, editors, *Arrays, Functional Languages, and Parallel Systems*, pages 93–106. Kluwer, Boston, 1991.

[FVH94] C. Foisy, J. Vachon, and G. Hains. DBML: de la semantique a l'implantation. *Journees Francophones des Languages Applicatifs*, February 1994.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. Conference on Principles of Programming Languages, January, 1988.

[GMN+94] A. Goldberg, P. Mills, L. Nyland, J. Prins, J. Reif, and J. Riely. Specification and development of parallel algorithms with the proteus system. Manuscript available through XMosaic from http://www.cs.unc.edu/proteus-publications.html, 1994.

[GS85] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[Hai93] G. Hains. Programmation fonctionelle et parallelisme: une approche pragmatique. manuscript, August 1993.

[HF93] G. Hains and C. Foisy. The data-parallel categorical abstract machine. In *PARLE'93: Conference on Parallel Architectures and Languages Europe*. Springer Verlag, 1993.

[HLA94] L. Huelsbergen, J. Larus, and A. Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, June 1994.

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

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

[KLS+91] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance FORTRAN Handbook*. The MIT Press, Cambridge, Massachusetts, 1991.

[Mac84] D. B. MacQueen. Modules for standard ML. In *ACM Symposium on Lisp and Functional Programming*, August 1984.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[MNP+91] P. H. Mulls, L. S. Nyland, J. F. Prins, J. H. Reif, and R. A. Wagner. Prototyping parallel and distributed programs in *Proteus*. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 26–34, December 1991.

[MT93] G. Morrisett and A. Tolmach. Procs and locks: A portable multiprocessing platform for standard ml of new jersey. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 1–12, San Diego, California, May 1993.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[PP93] J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA., May 19-22, 1993. ACM Press.

[Ske91] S. Skedzielewski. Sisal. In L. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–157. Addison-Wesley Publishing Co, 1991.

[Thi93] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide*, May 1993.

[TJ92] J.-P. Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3), July 1992.

[Wan80] M. Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, August 1980.