4-19-2013

Toward a Parallel Implementation of J: Data Parallelism in Functional, Array-Oriented Languages with Function Rank

Christopher Jenkins
Trinity University, cjenkin1@trinity.edu

Toward a Parallel Implementation of J: Data Parallelism in Functional, Array-Oriented Languages with Function Rank

Christopher Jenkins

Abstract

The notion of function rank has important implications for the field of parallel computing. In particular, certain formalizations of function rank can be helpful for exploiting potential concurrency in the form of data parallelism, because function rank can allow the programmer to express safely, easily, and in an automatically data parallel fashion both the application of a function to subcollections of a regularly shaped multidimensional collection and the extension of a function to similar problems in higher dimensions.

This paper illustrates this importance by discussing solutions using function rank to three parallel problems, each chosen to represent a different parallel design patterns. Additionally, included are both a set of proposals for a parallel implementation of J, which is an array-oriented, functional programming language with function rank, as well as a prototype implementation of a parallel regular collections library using function rank in Scala. Performance results for solutions using the prototype Scala library (as well as C with OpenMP, for comparison) to each of the problems are also given, though this work is intended only as a proof of concept.

Acknowledgments

Dr. Berna Massingill my advisor and friend, for agreeing to guide me through my research and share her expertise on parallel computing with me (as well as all the interesting tangents that came from our conversations).

Dr. John Howland for showing me the Tao of the J programming language, teaching me new paradigms for solving problems and sparking interest in a field of research I would never have discovered otherwise.

Dr. Victoria Aarons for being willing (and able) to wade through pages of technical writing to provide important feedback on structure and clarity in my thesis.

Drs. Mark Lewis, Seth Fogarty for agreeing to be on my thesis commitee and for answering obscure technical questions at only a moment's notice.

Valeri, Andrius, Vicky, Scott, Cinco, Faith for cheering me on during the final thesis boss-fight.

Nick for showing me the beauty of type theory, an extension to this work I might have never discovered, and for helping me work through difficult problems.

James, Joyce, Ross for being a loving family, supporting me throughout my entire time at Trinity

Nina my grandmother, who would have been proud to see me walk on the stage at graduation.

Toward a Parallel Implementation of J: Data Parallelism in Functional, Array-Oriented Languages with Function Rank

Christopher Jenkins

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

April 19, 2013

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Toward a Parallel Implementation of J: Data Parallelism in Functional, Array-Oriented Languages with Function Rank

Christopher Jenkins

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many scientific and business computing applications must work on large data sets naturally structured in regular, multidimensional collections. In order for these applications to achieve good performance, it is often the case that programmers must exploit any and all potential concurrency in the application through different approaches to parallel programming. One form of potential concurrency frequently encountered when programs operate on large collections of data is *data parallelism*, which is when programs update elements or subsections of collections in parallel. Many tools have already been developed to help programmers exploit data parallelism easily and safely (i.e., with reduced risk of race conditions or program crashes)[12][16]. However, many of these tools may not be as helpful for exploiting data parallelism on regular collections as they could be, and thus make it more difficult for programs to achieve good performance (that is, fast execution time), for two reasons.

First, many programming languages implement regular collections as nested collections;

e.g. a 2-dimensional, 2 by 3 array (or rank 2 array with shape 2 3) of integers would be implemented as an array of 2 arrays, with each of these containing 3 integers. However, in such languages irregular collections are also usually implemented as nested collections, with each of the sub-collections possibly containing a different number of elements from each other. Since both regular and irregular collections are implemented the same way, it can be difficult for programmers to view, manipulate, and verify any important properties of the structure of regular collections. For example, if a function requires that one or more of its arguments is a regular collection, programmers must either write code to ensure this requirement is met, increasing the difficulty of writing what could be an already complex program, or risk run-time indexing errors and program crashes.

Alternatively, regular collections can be implemented as vectors that have an associated shape vector. These shape-associated collections make the structure of a regular collection a field that can be viewed and manipulated and allow the programmer to have greater transparency and control when writing programs to operate on them. Unfortunately, most statically-typed languages do not have type systems advanced enough to capture all of the structural information of shape-associated collections statically, even though this is theoretically possible. However, some languages are able to capture some of this structure statically, enabling programmers to reduce or eliminate the need for manual verification, such as the fact that the arguments to a function are in fact regular collections, or that some collection satisfies a function's lower and/or upper bounds for rank[2][20][12]. Even when these properties of regular collections cannot be captured statically, making this information more transparent at run time should allow programmers to more easily verify that the required properties hold, reducing programmer effort and increasing program safety.

Second, in most imperative languages with regular collections, applying functions to elements or sub-collections of a collection means writing the function call within nested looping structures, typically a *for loop* Using this method to apply functions on these collections usually means that the number of dimensions (also called *data rank*) of the collection determines the number of loops required to do a specific operation. To put it another way, if an existing function that operates on regular collections needs to be extended to a collection of higher rank, or if the function only operates on scalar values but needs to be extended to some regular collection, the programmer usually must wrap the function in nested for loops. Without abstractions in the language or a library to deal with the general cases of such extensions, this activity is tedious in the trivial cases, adding unnecessary syntactic overhead to the programming process, and prone to error because programmers must instead write boilerplate code in order to accomplish it. Moreover, in some cases these extensions are data parallel, and can be done automatically and automatically exploit potential concurrency. However, using nested for loops obfuscates, and requires the programmer make changes to existing code in order to exploit, this inherent concurrency. Clearly it would be better for programmers to express the potential concurrency directly, and thus hopefully exploit potential concurrency directly, rather than expressing it as sequential for loop and either hoping a compiler can reverse this transformation or writing additional code to exploit the concurrency.

Functional programming languages, in contrast to imperative programming languages, allow programmers to view and write programs at a higher level of abstraction. When dealing with collections, this higher level of abstraction usually means that related types of operations can be expressed as *higher-order* functions like *map* and *reduce* that take functions as their arguments and return functions that operate on collections. These higher-order functions can allow programmers to more easily understand what operations are being done to regular collections than if the same operations were done imperatively using for loops because they capture the essence of what the operations *are*, more than how the operations are *implemented*. As a consequence, they also can allow programmers to see more easily where there is inherent data parallelism in the algorithm, such as all calls to *map*, or calls to *reduce* with associative operations, on large collections. The level of abstraction they provide programmers, as well as their lack of side-effects (an thus most sources of race conditions), make functional languages ideal for solving many kinds of parallel problems. However, since the higher order functions most functional languages provide for operating on collections are usually designed to operate on only one dimension at a time, they too must be nested in order to extend existing functions to collections of higher rank, with the same, though usually reduced, problem of unnecessary overhead and boilerplate code mentioned above.

Function rank, first introduced by K. Iverson in 1978[10] and implemented in the programming language J, is a functional abstraction that extends the notion of data rank to functions. Like other functional language abstractions, function rank can allow the programmer to directly express the application and extension of functions on specific ranks of regular collections through a higher-ordered function, called in J and in this paper the *rank operator*[6]. Expressing these extensions as a higher-order function makes it easier for a programmer to make them safely, quickly, and at a higher level of abstraction. In some cases, these extensions are so trivial that they can be done automatically, meaning the programmer need not modify the code at all[6] [8]. Furthermore, since multiple applications of the rank operator are equivalent to the cases where nested loops or nested higher-order function applications are inherently data parallel, it too is inherently data parallel. Consequently, languages with both formalized function rank and a rank-operator allow the programmer to exploit the inherent data parallelism of extending existing operations to collections of higher rank safely, quickly, and in some cases automatically.

However, currently the languages that meet these criteria, such as J, Sharp APL, and some others from the APL language family, are not in common use. One often-cited reason for this lack of use is that these languages are difficult to read, because, in order to use them effectively, a programmer must memorize dozens of 1 or 2 character functions each with different, sometimes unrelated use cases depending on whether the function takes one or two arguments[6][1]. These language design choices, however, are not required in order for a language to support function rank. A proof-of-concept may be needed to demonstrate that both the notions of function rank and associating collections with their shape are still very helpful for exploiting data parallelism on regular collections when available in a more modern, more popularly supported language.

The rest of the paper is organized as follows:

- The remaining sections of this chapter give the design plan (1.2) and implementation (1.3) of our research. Unfortunately, the latter did not quite fulfill the full scope of the former, so both a description of what was planned and what was actually accomplished within the time constraints are given.

- Chapter 2 gives the necessary background information for understanding this work, including a brief description of function rank and how it is equivalent to nested loops or nested calls to *map*, a discussion of relevant parallel design patterns for the example problem set, and a literature review of other attempts to solve the same or similar problems

- Chapter 3 proposes two new operators for future parallel implementations of the J programming language and illustrates how these operators might be used

- Chapter 4 gives the listing of our selection of example problems, giving briefly for each a description, a discussion of the relevant parallel design patterns for exploiting concurrency, and a description of how possibly to extend the problem to higher dimensions.

- Chapter 5 compares example solutions to the problems listed in Chapter 4 in J, Scala with Parallel-J, and C with OpenMP, and discusses the relative level of abstraction, scalability and performance of each solution.
- Chapter 6 presents our conclusions of this research and discusses future work.

In addition, appendices and a glossary of J functions used throughout the remaining chapters are given at the end of this paper, as aids to the reader.

1.2 Design Plan

The goal of this research was going to be to implement a parallel subset of the programming language J, called *Parallel-J*. Based on the documentation of J's current implementation[7], this language subset would require:

- a J language lexer and parser
- limited memory management
- a look-up table for predefined and user functions
- a read-evaluate-print-loop (REPL), and
- a subset of the J primitives, including:
  - global variable assignment
  - most array operations (creating, indexing, restructuring)

  - all of the basic arithmetic and logical operations
  - a limited selection of higher-ordered functions (composition, conditional, reduction)
  - J's function composition rules for sequences of functions (*trains*), and
  - the rank operator and function rank

The following language features would be excluded:

- namespaces (*locales*) and local variable assignment
- all existing environment-altering functions (the *foreign* operator, spelled `!:`)
- explicit scripts with imperative-style execution and control structure, and
- the J primitives which calculate complex algorithms (prime factorization, derivatives of polynomial equations, etc.)

These choices would allow solutions to the sample problems written in as simple and "idiomatic" J as possible, thus clearly showing the advantages to exploiting data parallelism when approaching these problems with function rank, without getting distracted with a discussion of J's more advanced or nuanced language features. Also to be included was both a proposal and an implementation of two new operators (parallel rank and parallel insert), and an extension to the existing *foreign* operator to allow for changing the parallel environment. The former would allow for the equivalent of parallel map and reduce operations at the specified rank(s); the latter would allow the programmer to set environmental parameters for parallelizing code, for example specifying the number of threads to use in executing a piece of code, or setting thread scheduling schemes. Although not entirely related to function rank, such environmental controls as would be included in the parallel environment are to be expected from any serious parallel computing language or library.

Scala, a programming language developed by Odersky et al.[15], was chosen to implement this parallel subset of J for the following reasons:

1. Scala supports multiple programming paradigms, such as
    - imperative, for the tasks of memory management and object creation,
    - object oriented, for more structured encapsulation of the rank associated with each function and data associated with arrays, and
    - functional, to more easily capture J's functional paradigm in the implementation and thus facilitate development
2. Scala has a feature-rich library for collections, including
    - support for many higher-order operations such as *map* and *reduce*, common to many solutions to data parallel problems[14], and
    - a library for parallel collections that exploits the concurrency in data parallel operations[17], which, being similar to the use-cases in this research, promised to make the parallel implementation relatively easy.

The performance of solutions written in Parallel-J to a suite of problems would be compared to performance of the same solutions written in C with OpenMP, J, and Scala. Also given for each problem would be a discussion of the relative level of abstraction, scalability to similar problems of higher dimension, and performance of each. It was not expected that Parallel-J would compare favorably to the raw performance of C or the current implementation of J; the hoped for result was to show that the performance Parallel-J scales relative to the number of threads, showing that this implementation does effectively exploit potential concurrency.

1.3 Implementation

Unfortunately, the given time constraints did not allow for such an ambitious project, leading to a change of focus. Rather than finishing a parallel subset of J, the J system libraries developed are instead used as a prototype for a Scala regular collections library that supports function rank. Taking the same set of parallel problems, this research shows both how this library allows for uniformly extending these problems into higher dimensions and how with future work it would be able to achieve good performance by automatically parallelizing solutions to problems in the given and in higher dimensions. Results are also compared with solutions written in C with OpenMP, along with a discussion of the relative level of abstraction, scalability, and performance of each.

Not all of the work done towards producing a parallel implementation of J applies to producing a Scala library with function rank. The majority of this work consists of the proposals for the *parallel rank* and *parallel insert* operators and the parallel environment library, given in Chapter 3, as well as a partial lexer for the J programming language, listed in Appendix A.

Chapter 2

Background

2.1 Function Rank

This section provides background information necessary to understand the advantages function rank has in exploiting data parallelism over other approaches.

2.1.1 History and Definition

Function rank was first developed and described by Kenneth Iverson in a series of research reports written at IBM[11][10]. In one such report, Iverson described it as "the most important notion needed to provide a simple and systematic basis for the uniform treatment of all 'mixed' or non-scalar functions"[11]. Since that time, the idea of function rank has matured and found its way into many dialects of APL, including J.

J's model of function rank is slightly different from what was first presented by Iverson[8][6]. The rank of a function `f` in J is a vector `v` of three values representing the data rank of `f`'s expected arguments. Since in J functions take either one or two arguments, the first value of `v` represents the expected data rank of `f`'s one-argument case (in J referred to as the *monadic* case); the second and third values in `v` represent the expected data rank of `f`'s two-argument (*dyadic*) case. If `f` has no restrictions on some or all of its arguments, this is represented in `v` with the value for infinity, spelled `_` ; if `f` operates on scalar values, this is represented as an entry in `v` of 0 (in J, scalars are collections of rank 0 and an empty shape vector).

Thus, for example, most arithmetic functions, such as addition, fundamentally operate on scalar values, and thus usually have the function rank `0 0 0`. These arithmetic functions, and all others which operate on scalar values, must be extended, whether manually by the programmer or automatically by the J environment, to operate on collections of rank $n \geq 1$. On the other hand, most collective operations, such as using an integer to index into collection (which has rank `0 _`), sorting a collection, and getting the shape of an array (both of which have rank `_`), operate on whole collections at once. In the above example, function ranks with only 2 elements represent the J primitive's dyadic use cases; similarly for the function ranks with only 1 element and J's monads.

2.1.2 Shape Agreement

In the trivial cases, where a function `f` is given arguments with ranks matching `f`'s function rank, J behaves much like any programming language without function rank.

```
   1 + 1
2
   show =: ] NB. Identity, used to display results
   integers =: i. NB. creates array with shape of argument
   NB.                populated with an incrementing value
   NB.                starting at 0
   show mat2_3 =: integers 2 3
0 1 2
3 4 5
   from =: { NB. Indexing into an array
   NB.           expressed as a function
   1 from mat2_3
3 4 5
```

In some cases, when the arguments to `f` do not match its function rank, `f` is automatically extended to the appropriate dimensions. For example, if `x` is a scalar, addition can always be extended so that `x` is added to every element of a collection `c` no matter `c`'s rank or shape.

```
   1 + mat2_3
1 2 3
4 5 6
```

In general, addition (and all other scalar functions) can be extended automatically over two regular collections `x` and `y` if the shape of one collection *prefixes* the other. This is called *prefix shape agreement*, or just *shape agreement*, and in this paper we will say when this happens that "the shapes of `x` and `y` agree under addition"[8]. We also say that this prefix is the *frame* for the two collections, and what remains of the shape vectors of `x` and `y` after dropping the prefix are their respective *cells*.

Going back to the above example: a scalar, a vector of 2 elements, and another 2 by 3 matrix will agree with `mat2_3` under addition, since the shapes of the scalar (empty shape), the vector, and the other matrix would prefix the shape of `mat2_3`; any collection of rank $n \geq 2$ whose shape begins with 2 3 will also agree with `mat2_3` under addition, since the shape of `mat2_3` would prefix its shape.

```
   100 200 + mat2_3
100 101 102
203 204 205
   NB. agreement: visualizes
   NB. how the cells of each
   NB. collection are paired with each other
   NB. before performing the desired operation
   agreement =: ; "

   NB. Show agreement of two collections above
   NB. under adition
   NB. The shape 2 is the frame;
   NB. the scalars are expanded to
   NB. vectors of 3 to match
   NB. the shape of mat2_3
   100 200 (+ agreement) mat2_3
+---+-+
|100|0|
+---+-+
|100|1|
+---+-+
|100|2|
+---+-+

+---+-+
|200|3|
+---+-+
|200|4|
+---+-+
|200|5|
+---+-+
```

Figure 2.1: Visualizing shape agreement, part 1

```
   mat2_3 + mat2_3
0 2  4
6 8 10
   NB. The frame is 2 3;
   NB. the scalar cells of both collections
   NB. are paired with each other
   mat2_3 (+ agreement) mat2_3
+-+-+
|0|0|
+-+-+
|1|1|
+-+-+
|2|2|
+-+-+

+-+-+
|3|3|
+-+-+
|4|4|
+-+-+
|5|5|
+-+-+
```

Figure 2.2: Visualizing shape agreement, part 2

```
   show arr2_3_2 =: integers 2 3 2
 0  1
 2  3
 4  5

 6  7
 8  9
10 11
   arr2_3_2 + mat2_3
 0  1
 3  4
 6  7

 9 10
12 13
15 16
```

Figure 2.3: Visualizing shape agreement, part 3. Continues in Fig 2.4

In this last example in Figure 2.3 and Figure 2.4, every scalar of `mat2_3` was added to both scalars of each vector in `arr2_3_2`. Another way to conceptualize this is that J made an implicit *map* on the scalar elements of `mat2_3`, expanding each into a vector of 2 scalars. J could do this because, with a *frame* of `2 3`, `mat2_3`'s *cells* were scalars and `arr2_3_2`'s *cells* were vectors of two scalars.

2.1.3 Rank Operator

With what we have developed so far, we are still unable to perform the operation of adding a vector `vec3` of 3 scalars to `mat2_3`, since shapes 3 and 2 3 have no non-empty prefix.

```
   show vec3 =: i. 3
0 1 2
  vec3 + mat2_3
|length error
|   vec3    +mat2_3
```

```
   NB. The frame is 2 3;
   NB. The scalar cells of mat2_3
   NB. are expanded to vectors of 2
   NB. to match the shape of
   NB. arr2_3_2
   arr2_3_2 + agreement mat2_3
+--+-+
|0 |0|
+--+-+
|1 |0|
+--+-+

+--+-+
|2 |1|
+--+-+
|3 |1|
+--+-+

+--+-+
|4 |2|
+--+-+
|5 |2|
+--+-+


+--+-+
|6 |3|
+--+-+
|7 |3|
+--+-+

+--+-+
|8 |4|
+--+-+
|9 |4|
+--+-+

+--+-+
|10|5|
+--+-+
|11|5|
+--+-+
```

Figure 2.4: Visualizing shape agreement, part 4

However, it should be possible to add `vec3` to each of the vectors of `mat2_3`, since each vector has the same length. In order to accomplish this and many similar use cases where the desired extension of a function is not the default extension, J also includes a *rank operator* (which is spelled with double quotes: `"`). The rank operator is a higher order function which takes as its first argument a function (not higher-ordered) and as its second argument a vector of 1, 2, or 3 numeric values, and returns a function which performs the same operation as the argument function but on the specified data rank[8].

Therefore, the following command,

```
   vec3 +"1 mat2_3
0 2 4
3 5 7
```

is read "add the rank 1 items of `vec3` to the rank 1 items of `mat2_3`." In terms of *frames*, *cells*, and implicit *maps*, we say that the frames of the rank 1 items in `mat2_3` and `vec3` are 2 and an empty frame, respectively, and they share a common cell size of 3. Because the frame at rank 1 of `vec3` prefixes the same frame `mat2_3` (the empty frame prefixes every frame), the shapes now agree, and there is an implicit map on the single vector element of `vec3` expanding to a matrix of two vectors.

2.1.4 The Application of a Function with Rank on its Arguments

For any function `f` with function rank `r` and arguments `x` and `y`, the following steps give a high level description of how `f` is applied to its arguments[8].

1. Calculate the cell shape at rank `r` of `x` and `y` by taking the `r` smallest dimensions of the shape vector of each. E.g., the cell shape of `mat2_3` at rank 1 is 3, since each vector contains 3 items.

2. Calculate the frame shape at rank `r` of `x` and `y` by removing from the shape vector of each their respective cell shapes. E.g, at rank 1, the frame shape of `mat2_3` is 2, the frame shape of `arr2_3_2` is 2 3, and the frame shape of `vec3` is an empty frame.

3. If the frame shape of `x` and `y` do not agree, then return with an error. Otherwise, extend the argument with the smaller frame shape via an implicit map on its cells at rank `r`. If the frame shape of `x` and `y` are the same, do nothing.

4. Apply `f` to every cell at rank `r` of `x` and `y`. If `f` is a user defined function `u` with function rank `ru` given with the rank operator, (i.e., `f =: u ("ru)`) repeat this process with each of the cells of `x` and `y`, `u`, and `ru`

5. Reassemble the result cells of the previous step using the agreed frame shape.

2.1.5 Inherent Data Parallelism

While quite a few of these steps have some exploitable concurrency, step 4 has the most potential for performance increases through parallelism. It is inherently data parallel because each of the cells of x and y are operated on completely independently of each other. For large computations, it is also the most computationally intensive because not only is this step itself recursive, but also because these cells can themselves be large regular collections.

Finally, one consequence of having this common set of steps for applying all functions with function rank is that all such functions can be parallelized in the library code. This means that applications which uses a parallelized function rank library can automatically exploit the inherent concurrency of their problem, provided this problem can be expressed naturally in terms of function rank.

2.2 Other Approaches

2.2.1 Regular Parallel Arrays in Haskel

In 2010, Keller et al. published a paper[12] describing work they had done creating a Haskell library, which they named *Repa*, that implements and parallelizes regular arrays. There is much to commend about their work, including "that it (1) is purely functional, (2) supports reuse through shape polymorphism, (3) avoids unnecessary intermediate structures rather than relying on subsequent loop fusion, and (4) supports transparent parallelization," and that it is a library for a functional language with relatively wide use. There are two features of *Repa* specifically that influenced work on Parallel-J; these are *index functions* and static capture of a collection's shape information.

In order to achieve good performance on functions that fall into the family of operations known as *index transformations*, such as transposing or shifting a rank 2 array, *Repa* formalizes the notion of an *index function*. An *index transformation* is an operation on a regular collection that conceptually changes how the collection is indexed. For example, using a C-style notation, transposition might be represented as `transpose2D(matrix)[i][j]` $\Leftrightarrow$ `matrix[j][i]`, for all integers i, j. *Index functions* allow index transformations to be implemented purely as a change to how a given collection is indexed, rather than by allocating new memory for the transposed collection and copying every element. Parallel-J currently lacks this feature, but a future Scala regular collections library could easily support index functions.

Impressively, *Repa* also statically captures some of the effects functions have on the shapes of their arguments. For example, the type signature of *sum* shows that, given a numeric array of rank n, it returns an array of rank $n-1$ which has the same shape as the argument array except for the rightmost (smallest) dimension. Another way to state this is, their sum library function takes an arbitrary array of rank $n \geq 0$ and sums that array's vector elements, returning an array whose shape is the shape of the argument array with the smallest dimension dropped off. Consequently, passing *sum* a rank 0 array is a compile-time error. In general, this library "enables [the user] to track the rank of each array in its type, guaranteeing the absence of rank-related runtime errors"[12].

Repa's static capturing of the rank of a function's arguments is equivalent to J's notion of function rank. It is implemented as a list of the type *Int* and uses Haskell's pattern-matching capabilities and some language extensions to analyze the structure of this list. However, unlike J, Repa appears to lack a rank operator. Instead, functions must be extended manually in order to operate on arrays of higher rank. While *Repa*'s manual extensions are safer from error than extending through nested *maps* or *for loops* due to its type safety and due to reducing some of the boilerplate code of the latter in functions like *replicate* and *backpermute*, because it does not express these extensions as higher-ordered functions, programmers using *Repa* must still work at a conceptually lower-level of abstraction, and still must write more boilerplate code, than using J's rank operator would require of them.

For example, while in J the idiom for *sum* (spelled `+/`) automatically operates on the rank $n-1$ items of a rank n array, in Repa, *sum* by default operates only on scalar values in each vector of a collection. In order to scale the existing *sum* function in Repa to any array of rank $n > 1$, a new function must be written for each dimension which manually extends *sum*[12]. In contrast, in order to get the same behavior in J of Repa's *sum* function, no manual extension is required; it is `+/ " 1`, which in English reads rather intuitively as "apply sum to the vector elements of its argument."

Finally, although both J and Repa support some notion of function rank, it seems conceptually easier to understand, for example, that `f (" 3)` means "apply f on the rank 3 items", than it is to understand Repa's equivalent, `(sh.:Int.:Int.:Int)` as it would appear in the function declaration below:

```
f ::  (Shape sh, Elt e) => Array (sh :.  Int :.  Int :.  Int) e -> Array sh e
```

Reap's greater verbosity is partially due to Haskell being statically typed, which is what also allows *Repa* to capture the effects operations have on data rank at compile time. J, in contrast, is dynamically typed, and while this means J does not provide any mechanisms for catching errors before runtime, its functions also do not need to statically document their effects on data, simplifying the way programmers write and use code.

2.2.2 SA-C, Boost MultiArray

While we believe that *Repa* is the best of the solutions we have found so far, due to being already parallelized, capturing the effects functions have on the rank of their data, and reducing boilerplate code, it is appropriate to mention other influential work in the subject of data parallelism on regular collections. Our analysis is mostly in agreement with the developers of *Repa*[12].

Single-Assignment C (*SA-C*) is a functional, C-like language that has many of the same advantages as *Repa*[12][20]. Unfortunately, this also means that it has the same limitations, most notably the lack of a rank operator. Additionally, unlike *Repa* and our own research (but like J), it is a special purpose array programming language and not an extension to an existing, general purpose programming with a broad user base and with access to large and well developed libraries.

In contrast, the C++ library Boost.MultiArray is a library for a general purpose and widely used programming language[2]. However, its ability to analyze and operate on the structure of regular collections is limited compared to SA-C or *Repa.* Furthermore, it does not benefit from a naturally parallel implementation, its arrays are operated on in a conceptually lower (imperative) level, and it also does not have an equivalent to the rank operator.

Chapter 3

Proposed Operators for a Parallel Implementation of J

3.1 Rationale

Ideally, using a parallel library would automatically reduce a program's runtime without requiring the programmer to write a single line of parallelizing code. However, this is often not the case. Frequently, programmers must use domain knowledge of the problem or platform to achieve good run-time results.

In order to grant this flexibility in future parallel implementations of J, we propose introducing two new operators, called the *parallel rank operator* and the *parallel insert operator*. The *parallel rank operator*, described in Section 3.2, would allow the programmer to specify the ranks on which to parallelize code. The *parallel insert operator*, described in Section 3.3, would allow the programmer to parallelize reduction operations with associative functions. Additionally, we propose a new system library that would allow the programmer to give annotations or force changes in the underlying parallel environment, described in

Section 3.4

The spellings `"::` and `/::` for the two parallel operators were chosen for mnemonic and compliance reasons. Mnemonically, each uses the same base character as their sequential analogs, (`"` and `/`), as well as two “parallel” colons, making it easier to remember their functions. They also require no changes be made to the existing J lexer[7], as demonstrated below:

```
   NB. The J Lexer as a primitive
   NB. Takes a character string and
   NB. puts each lexeme in a ‘box’
   jlexer =: ;:
   jlexer ’+/::("1) mat2_3 +("::1)("2) arr2_2_3’
+-+---+-+-+-+-+------+-+-+---+-+-+-+-+-+--------+
|+|/::|(|"|1|)|mat2_3|+|(|"::|1|)|(|"|2|)|arr2_2_3|
+-+---+-+-+-+-+------+-+-+---+-+-+-+-+-+--------+
```

For the purposes of this discussion, the important result the above example demonstrates is that the string of characters `"::` is read as a whole lexeme, and is placed entirely in a box, rather than being interpreted as several lexemes, which would result in the characters being split into several boxes. For more detail on the behavior of the J language, consult Appendix A.

For the parallel environment library, it seemed best to augment the existing “foreign” operator (spelled `!:`). The foreign operator was designed to allow the programmer to change environmental parameters such as print precision, file I/O, etc[6], and so fits conceptually with the idea of changing the state of a parallel environment. We chose 111 as the numeric encoding for the parallel environment library, again for mnemonic reasons: 111 looks like three parallel lines (11 was already taken).

3.2 Parallel Rank Operator: `"::`

3.2.1 Usage

The proposed parallel rank operator is a conjunction whose first argument is a function and whose second argument is the ranks to both apply the function and to parallelize it. It would be functionally equivalent to the rank operator, i.e.

`f (":: r) y` $\Leftrightarrow$ `f (" r) y`

and

`x f (":: r) y` $\Leftrightarrow$ `x f (" r) y`

for all f, r, s, and y

Its purpose would be to override other parallel system defaults for automatic parallelization to guarantee that the resulting function would create parallelizable tasks from the operations on sub-arrays of the given rank that would then be distributed to the available threads. To illustrate, consider the examples in Fig 3.1. Incrementing numeric values always applies to scalars, so the function `increment` always gives the same result, regardless of the rank of the sub-collections to which it is applied. Using the parallel rank operator, the behavior increment of always having the same result regardless of the rank applied would remain; however, as the last example visually illustrates, specifying `increment (":: 1)` would result in the environment parallelizing the operation of incrementing each of the vector elements of `mat2_3`. In this example, a programmer could create parallelizable tasks from incrementing the six scalars, the two vectors, and the one matrix.

```
   NB. increment: increments the values of a numeric array
   increment =: >:
   increment mat2_3
1 2 3
4 5 6
   NB. Link places both its arguments in ‘boxes’
   NB. This demonstrates that incrementing a collection
   NB. is the same regardless of the rank applied
   (increment"0 link increment"1 (link =: ;) increment"2) mat2_3
+-----+-----+-----+
|1 2 3|1 2 3|1 2 3|
|4 5 6|4 5 6|4 5 6|
+-----+-----+-----+
   NB. showTasks visualizes how the subcollections of mat2_3
   NB. would be broken up into tasks that would then be
   NB. concurrently operated on ny threads
   showTasks =: agreement (< @)
      (increment (" 1) showTasks) mat2_3
+-----+-----+
|0 1 2|3 4 5|
+-----+-----+
```

Figure 3.1: Visualizing tasks created by the parallel rank operator

A more complex example involving functions of two arguments, as well as repeated applications of the rank operator, is given in Fig 3.2. Using the parallel rank operator, the line of code `mat2_3 +("::2) arr2_2_3` would create parallelizable tasks for adding the elements of each of the two matrices, whereas `mat2_3 +("::1)("2) arr2_2_3` would create parallelizable tasks for adding the elements of each of the four vectors.

Some open questions remain, most notably the behavior of the parallel environment when multiple applications of the parallel rank operator are used. For example, should multiple applications of the parallel rank operator result in multiple levels of parallelizable task creation, leading to increased overhead from thread creation and scheduling, or should the outermost application determine the only level at which parallelizable tasks are created,

```
   show arr2_2_3 =: integers 2 2 3
0  1  2
3  4  5

6  7  8
9 10 11
   mat2_3 +"2 arr2_2_3
0  2  4
6  8 10

6  8 10
12 14 16
   mat2_3 +("1)("2) arr2_2_3
0  2  4
6  8 10

 6  8 10
12 14 16
   mat2_3 (+("2) showTasks) arr2_2_3
+-------------+---------------+
|+-----+-----+|+-----+-------+|
||0 1 2|0 1 2|||0 1 2|6  7  8||
||3 4 5|3 4 5|||3 4 5|9 10 11||
|+-----+-----+|+-----+-------+|
+-------------+---------------+
   mat2_3 (+("1) showTasks) arr2_2_3
+-------------+---------------+
|+-----+-----+|+-----+-----+  |
||0 1 2|0 1 2|||3 4 5|3 4 5|  |
|+-----+-----+|+-----+-----+  |
+-------------+---------------+
|+-----+-----+|+-----+-------+|
||0 1 2|6 7 8|||3 4 5|9 10 11||
|+-----+-----+|+-----+-------+|
+-------------+---------------+
```

Figure 3.2: Visualizing tasks created by the parallel rank operator for two collections

leading to unparallelizing previously parallelized programs when they are used in larger functions? Either possibility could have significant impact on program run time. One solution may be to allow the programmer to chose which behavior best suites their needs using the parallel environment libary, described in Section 3.4.

3.3 Parallel Insert Operator: `/::`

3.3.1 Rationale

Frequently, programmers need to perform some sort of reduction operation on an entire collection, for example finding the sum, product, maximum, minimum, etc. of a collection of numbers. It's well known that when the reducing operation f is associative, i.e. when $f(x, y) = f(y, x)$ for all f, x, y (and for a sufficiently tolerant comparison, when dealing with floating point values), then the reduction can be carried out in a parallel fashion with little effort on the programmer's part. Therefore, a future parallel implementation of J should allow the programmer to explicitly state that a specific associative function should be used to reduce an array in parallel, since in the general case the default insert operator does not assume its argument function is associative, nor is it immediately obvious how to determine if a user-defined function is associative.

3.3.2 Usage

As a unary, higher ordered function (*adverb*), the parallel insert operator would be equivalent to the sequential insert operator for all associative operations. Thus, the J session depicted in Fig 3.3 would have the same results for every line of code executed, regardless of whether `insert` was assigned to be `/::` or `/`

An open question remains about how the parallel insert operator should behave on

```
   insert =: /
   show vec10 =: integers 10
0 1 2 3 4 5 6 7 8 9
   + insert vec10
45
   NB. greaterOf takes two numbers
   NB. and returns the greater of them
   100 (greaterOf =: >.) 10
100
   greaterOf insert vec10
9
   + insert mat2_3
3 5 7
   NB. Flatten takes any array
   NB. and converts it to a vector
   + insert (flatten =: ,) mat2_3
15
   NB. sum each of the vectors
   NB. in mat2_3
   + insert ("1) mat2_3
3 12
```

Figure 3.3: Using the parallel insert operator

non-associative functions. Without associativity, the results of reducing an array with a function depend on the order by which the function is applied, destroying the possibility of concurrent operations. One possible way to deal with non-associativity is to define the domain of parallel insert to be only the associative primitives. This would prevent logical errors such as radically different results for the same calculations on the same data, depending on the thread scheduling scheme. However, this approach would likely require adding an additional table of information that would store whether or not the given primitive was associative. This approach would also exclude the possibility of user-defined functions that are associative from benefiting from parallelization.

Another possibility would be ignore the question of whether or not a function `f` is associative, and try to parallelize `f` as if it were associative. Using this approach could lead to errors that might be difficult for a novice parallel programmer to debug and may even go unnoticed, since those errors would be at the logic-level and not during run-time. Finally, a middle solution could be taken in which the programmer chooses to switch between these approaches using the parallel environment library, discussed below.

3.4 Parallel Environment Library: `111 !:`

3.4.1 Conventions of the Foreign Operator

The foreign operator is a two-argument higher order function, or in J a *conjunction*, whose arguments are always numeric. Conceptually, these arguments serve as an index into system libraries and functions. The left argument indexes the desired library. E.g., 2 indexes the library for functions which affect the host machine, 9 indexes the library for viewing and setting global J parameters (such as print precision), and etc. The second argument indexes a specific function. For example, `9!:6` is the function which displays the print characters for J's box type.

3.4.2 Usage

There are many options to consider for implementing a full suite of functions for a parallel environment library. A more full review of the functionality of other shared-memory parallel libraries, such as OpenMP[16], is required to make sure such a library provides all the functionality parallel programmers expect. However, two types of functionality would almost certainly be required. They are:

1. Getting and setting the total number of threads available in the parallel environment,

```
   (111 !: 0) '' NB. Get number of threads.
1
   (111 !: 1) 4 NB. Set number of available threads to 4.

   (111 !: 2) '' NB. Get numeric encoding of thread scheduling scheme.
0
   (111 !: 3) 1 NB. Set thread scheduling to, for example, round robin.

   (111 !: 4) '' NB. Get parallel rank nested thread creation flag
0
   (111 !: 5) 1 NB. Enable nested thread creation

   (111 !: 6) '' NB. Get parallel insert restriction flag
0
   (111 !: 7) 1 NB. Enable restrictions on insert

   +/::("1) mat2_3 +("::1)("2) arr2_2_3
 6 24
24 42
```

Figure 3.4: A hypothetical J session using the parallel environment library

with an extension for J's value for infinity to mean no user-specified limit on the number of threads, and

2. Getting and setting the thread scheduling schemes (such as static, round-robin, etc.). These would be encoded as numeric values, in keeping with the conventions of the foreign libraries.

Additionally, as discussed in Sections 3.4 and 3.3, it may be desirable to change the default behavior of the parallel rank operator and/or parallel insert operator, in order to parallelize on user-defined associative operators.

The example given in Fig 3.4 is not based on any existing J REPL, but an illustration of how programmers might use this proposed library. The last line would parallelize addition on the row elements of `mat2_3` and `arr2_2_3`, then parallelize the sum operation on each of the vectors of the resulting array, using 4 threads with a round-robin scheduling scheme.

Chapter 4

Example Parallel Problems

4.1 Rationale

The solutions to each of the following problems illustrate how the notion function rank both provides advantages in approaching different forms of exploitable concurrency, as well as allows for extensions to similar problems in higher dimensions easily, safely, and often automatically. For each of these problems, there is a trivial extension into a higher dimension: gather several rank n inputs to the problem into a rank $n+1$ collection and use the parallel rank operator to evaluate each of these inputs concurrently. By specifying that the original function operates on rank n elements, this extension is done automatically. However, each section also discusses, when appropriate, the usefulness of function rank when the extension is more complicated. Solutions for each of the problems in the following sections using the partial implementation of the Parallel-J system are given in Appendix C.

To demonstrate the usefulness of function rank, the problems were chosen so as to illustrate different kinds of *parallel design patterns*. *Design Patterns* are general, reusable solutions to a common class of problems. Software design patterns were first made popular for object-oriented programming through the work of the "Gang of Four"[4], and are now seeing interest for applications in parallel computing[13]. This research has chosen to focus on the pattern language developed by Mattson et al. presented in their book, *Patterns for Parallel Programming*[13]. Using parallel design patterns to guide the choices and discussion of these problems should allow this limited selection of problems to better represent families of parallel problems.

4.2 Calculating Pi with Numerical Integration

4.2.1 Description

One method of calculating π is to find the area under the curve of the equation $frac11 + x^2$ on the interval $[0, 1]$. This can be done with the integral in Eq 4.1.

$$\pi = \int_0^1 \frac{4}{1 + x^2}\, dx \tag{4.1}$$

One way to approximate this integral is using a left-handed Riemann sum. This method works by dividing the desired interval into n sub intervals, evaluating the function of concern at the left hand side of the interval, called x_i (where $0 \leq i < n$), and approximates the are underneath the curve by finding the area of a rectangle whose width is the width of the interval and height the value of the function at x_i.

$$\frac{1}{n} \cdot \sum_{i=0}^{n} \frac{4}{1 + x^2}, x = \frac{i}{n}$$

4.2.2 Relevant Design Pattern

This problem was chosen to represent the *Task Parallelism* pattern, in which "the problem is decomposed into a collection of tasks that can execute concurrently"[13]. In this problem, the concurrent tasks are calculating $\frac{4}{1+x^2}$ for the different values of x. There is also a task dependency: the multiplication of $\frac{1}{n}$ and the sum of the values produced by each task cannot occur until all tasks have finished. This task, which is a reduction on addition, can also be executed in parallel, with sub-tasks summing the resulting areas concurrently. Each individual task of calculating the area of a rectangle takes roughly the same amount of time, as well, meaning no sophisticated scheduling is required in order to exploit concurrency effectively for large n.

4.2.3 Solutions

C with OpenMP

A parallel solution to this problem using C with the OpenMP libraries is given below. It is based on the solution given in Appendix A of Mattson et al.[13].

```
#include <stdio.h>
#include <omp.h>
#define NUM_STEPS 100000000

int main(int argc, char *argv[]) {

    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) NUM_STEPS;

    #pragma omp parallel for private(x) reduction(+:sum)
    for (int i=NUM_STEPS-1; i >= 0; --i) {
        x = i*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf("pi %lf\n", pi);
    return 0;
}
```

This solution parallelizes the for loop; i.e., it distributes iterations of the loop among threads to execute concurrently. The program specifies both that `x` is a temporary work variable and so should not be shared among threads, and that there is to be a reduction on the variable `sum` with addition. Since the entire body of the for loop is parallelized, including the update of `sum`, the annotations notifying the compiler of the reduction and private work variable is critical; otherwise, there could very well be race conditions.

J

A sequential J solution to this problem is given below, with some definitions to facilitate a discussion of the behavior of a solution in the proposed Parallel-J. The operators `p_rank` and `p_insert` are defined as the normal rank and insert operators, respectively, but are included to demonstrate where parallelism would be exploited in a parallel implementation of J.

```
p_rank   =: " NB. Would be "::
p_insert =: / NB. Would be /::
integers =: i.

func =: (4 (div =: %) (1 + (square =: *:))) p_rank 0
xs =: integers div (n =: ])
num_int_pi =: (sum =: + p_insert)@:func@xs div n
```

This solution works conceptually by taking the for loop from the previous C solution and expanding it into an array, generated by `xs`. Operations which were done in the body of the C for loop are instead expressed as operations on the whole array. Because J is functional, and because it allows the programmer to express collective rather than item-by-item operations, the need for the temporary work variable `x` used in the previous solution, a potential source of race conditions and thus error, is eliminated.

Similarly, whereas the C solution required protecting against race conditions when reducing on the `sum` variable by giving compiler annotations, in the J solution race conditions could be avoided altogether. Either the `sum` operation would be safely parallel with `:` or it would be sequential with `/`.

A solution using the Parallel-J system which takes exactly the same approach as the J program can be found in Section C.1.

4.2.4 Extensions

The problem of calculating π by finding the area under a specific curve generalizes to calculating any useful value by finding the value of an integral. In general, the domain over which an integral takes place need not be 1-dimensional, as it was in this example (over the closed interval $[0, 1]$). Instead, the desired value may most naturally be expressed as a volume or hyper-volume. For this example problem, the extension to higher dimensions would be calculating π by finding a volume or hyper-volume.

The following discussion will focus on generalizing the creation of the set of points in n-dimensional space over the interval $[0, 1]^n$. The choice of function f to apply to the points in n-space is entirely based upon the value to be calculated, and while it is possible to extend the generation of points in n-space to arbitrary intervals, doing so would only complicate the example without significant insight into the use of the rank operator.

The most immediate extension of the C solution to any specific rank n would be to wrap the program in the appropriate number of for loops. In C, the function f would probably take both an array of coordinate values as well as a parameter `size` specifying the current dimension, so the programmer would also have to make sure to change this parameter in

```
   xs =: (n (base=:#:) integers@:(* insert)) div("1) n

   NB. Divide [0,1]x[0,1] into 4x4 grid
   xs 4 4
   0    0
   0 0.25
   0  0.5
   0 0.75
0.25    0
0.25 0.25
0.25  0.5
0.25 0.75
 0.5    0
 0.5 0.25
 0.5  0.5
 0.5 0.75
0.75    0
0.75 0.25
0.75  0.5
0.75 0.75
   xs 10
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
```

Figure 4.1: Extending the creation of coordinates to arbitrary rank

each version for a new dimension.

While this extension only involves changing two parts of the code, it is tedious and must be done for every such extension. On the other hand, finding a general solution could be somewhat involved as it is perhaps not clear at first sight how to begin such an approach. The solution to this aspect of the problem along with some example uses is given in J in Fig 4.1:

The function `xs` now takes a vector `vn` of length n specifying how to break up the interval $[0, 1]^n$. To calculate the appropriate coordinates to do the Riemann sum, the function first calculates the total number of points required (reducing `vn` with multiplication). Then it finds, for each of the integers i between 0 and one less than this total, the array representation of i in base `vn`. Finally, like the above example, xs divides each of these values by the original input so that the resulting points are in the appropriate bounds. (For more on the behavior of the function `base`, see Appendix E).

Because of the way xs is defined, no matter the dimension on which the programmer desires to calculate their integral, the points for applying the desired function can always be represented by at most a rank 2 array

```
shapeOf xs vn;⇔;(* insert vn), n
```

To approximate the desired integral, apply the appropriate function (in an inherently parallel fashion) to each of the vector elements (which represent the coordinates to calculate the Riemann sum), then multiply the sum of the results by the appropriate value (`1 div (* insert vn)`).

4.3 Conway's "Game of Life"

4.3.1 Description

Conway's "Game of Life"[5] is a cellular automaton which is "played" on an infinite or bounded 2-dimensional grid. Every location on this grid represents a cell, which is either alive or dead. At each iteration, every cell is updated by applying the following set of rules:

1. If the chosen cell is dead, then
   (a) If this cell has exactly 3 neighbors that are alive, this cell becomes alive on the next iteration
   (b) Otherwise, the cell remains dead

2. If the chosen cell is alive, then
    (a) If this cell has either 2 or 3 neighbors that are alive, this cell remains alive on the next iteration
    (b) Otherwise, this cell becomes dead on the next iteration

It is often convenient to represent each cell with a numeric value that is 0 when the cell is dead, and 1 when the cell is alive. Then, to calculate the number of alive neighbors for each give cell, one need only sum the values of all of the neighbors.

4.3.2 Relevant Design Patterns

This problem represents the *Geometric Decomposition* design pattern,[13] in which "the algorithm [is] organized around a data structure that has been decomposed into concurrently updatable 'chunks'." For the bounded version of this problem, the data structure is a regular rank 2 array with shape *n, m*, and the concurrently updatable chunk can be any subregion of the grid, with the limiting case that each cell is a chunk. While there are no data dependencies in this problem, care must be taken when using an imperative approach, as cells must be updated by reading their neighbors' current, and not future, values.

4.3.3 Solutions

C with OpenMP

A fragment of a parallel solution in C with OpenMP for the Game of Life is given below. The solution was work done by the author for an undergraduate course in parallel computing. The fragment contains the function which updates each cell, which is where most of the computational concurrency for the game of life is found. Notice the two arguments to the function, `board` and `new_board`. In the main program loop (not shown), these parameters are swapped every iteration, avoiding the potential race condition mentioned above.

```
#include <stdlib.h>
#include <omp.h>

/* data structure for two-dimensional array */
typedef struct twoD_array {
    int rows;
    int cols;
    int ** elems;
} twoD_array_t;

void update_board(twoD_array_t *board, twoD_array_t *new_board);

/*
 * updates board configuration
 */
void update_board(twoD_array_t *board, twoD_array_t *new_board) {
    int i, j;

    #pragma omp parallel for private(j) schedule(static)
    for (i = 1; i <= board->rows-2; ++i) {
    for (j = 1; j <= board->cols-2; ++j) {
        int neighbs = 0;
        int mycell = board->elems[i][j];
        int k, l;

    /*count neighbors*/
    for (k = i - 1; k <= i+1; ++k) {
     for (l = j - 1; l <= j+1; ++l) {
            if (!(k == i && l == j) ) {
                neighbs += board->elems[k][l];
            }
        }
        }

    /*Logic of game*/
    if (mycell) {
            if (!(neighbs == 2 || neighbs == 3))
                mycell = 0;
    }

        else {
            if (neighbs == 3)
                mycell = 1;
        }

    /*Update board*/
        new_board->elems[i][j] = mycell;
    }
    }
}
```

In the code given above, the tasks given to threads are the iterations of the outermost for loop, i.e. the operations on each of the row elements, indexed by i. Again, we see that a safeguard for the index j is required in order to avoid race conditions. This safeguard, as well as the swapping back and forth of grids (which would be required in a sequential solution similar to this), are changes in code which have to be made not because of the nature of the computation itself, but because of the imperative, low-level nature of the tools to express it.

J

Like above, the J code below includes only the fragment responsible for updating the cells of the Game of Life.

```
p_rank =: "
p_insert =: /

NB. No neighbors at edge
shift =: |. !. 0

NB. Specifies neighbors
NB. D=Down, U=Up, R=Right, L=Left
NB.              UL       U      UR    L      R     DL     D     DR
shiftBy =: 8 2 $ _1 _1   _1 0   _1 1   0 _1   0 1   1 _1   1 0   1 1

NB. Generates higher-rank array for counting neighbors
neighborArray =: shiftBy&(shift " 1 _)
```

```
   NB. Reshape: binds a shape vector to a rank 1 array
   reshape =: $
   show board =: 3 3 reshape 0 1 1  0 1 0  1 0 0
0 1 1
0 1 0
1 0 0
   NB. The up-left, up, and up-right neighbors
   0 1 2 from neighborArray board
0 0 0
0 0 1
0 0 1

0 0 0
0 1 1
0 1 0

0 0 0
1 1 0
1 0 0
```

Figure 4.2: Neighbors of cells in the Game of Life expressed as a 3D array

```
NB. Sums the neighbor values, finding how many are alive
listNeighbors =: (+ p_insert)@: neighborArray

NB. Rules of the game, as an array
NB.      0 (cell dead)   1 (cell alive)
rules =: (3 = ])       ‘ ([: +./ 2 3 = ])

NB. Uses cell state to index into rule to apply
NB. appliedBy =: @.
nextState =: rules @. (cell =: [)("0)(p_rank 1) listNeighbors
```

One important difference between this solution and the OpenMP version is the approach to calculating the number of neighbor cells that are alive. Instead of using array indexing at an item by item level, the neighbors are represented as a rank 3 array of shifted versions of the original board. In order to represent that cells near the edges do not have neighbors, 0s are shifted in the appropriate places.

Calculating the number of live neighbors of a cell by using a rank 3 array is a data parallel approach to the problem, since now the sum acts on every cell in the same position in every rank 2 item. This approach is a safer alternative to requiring that the location of the cell is known to evaluate its neighbors, which potentially causes race conditions on thread-local variables. Similarly, instead of specifying a reduction on a work variable `sum`, the parallel insert operator parallelize the operation in an automatically safe fashion.

One objection to this approach would be the potentially dramatic increase of memory use in the J approach compared to C with OpenMP. This increase in memory use can be avoided by using *index functions*, described in Section 2.2.1. With an index function, only one copy of the original rank 2 would be kept, and instead of direct indexing into a rank 3 array, an index function is used to determine what the direct index into the rank 2 array would be.

A solution using the Parallel-J system which takes a similar approach as the J program (but includes a non-J for loop for the iteration) can be found in Section C.2.

4.3.4 Extensions

Of the listed problems, the Game of Life perhaps best demonstrates the utility of generalizing the application of functions to higher rank arrays using function rank, not only because the problem is expressed in a data parallel fashion; in general, simulations of cellular automata need not be restricted to two dimensions, but can occur on arbitrary dimensions. This discussion of extending the Game of Life to higher dimensions will focus on calculating the number of neighbors in any dimension. While it is possibly desirable that the rules governing the life of a cell would also change depending on the dimension, the particular set of rules used is omitted from the discussion because the decision of which rules govern the life of a cell is dependent on the results the programmer wishes to achieve, and not the general method for achieving them.

In the C solution with OpenMP, the obvious extension of the problem to any specific rank is to nest the existing for loops with additional for loops. Since for loops are used for both examining each cell, as well as examining each neighbor of the given cell, this involves writing 2 for loops with 2 extra variables for every additional dimension desired. Additionally, if the convenience of the C structure `twoD_array` is desired, the structure must be rewritten for every rank desired. Again, on the other hand, it is may not be easy for a programmer to see how to begin an approach which handles the problem in the general case. The solution to the general approach in J is given below.

```
NB. General algorithm for finding
NB. shifts of board of arbitrary rank
decr =: <: NB. decrement
integers =: i.
dim =: #@$
base =: #:
n =: ]
copies =: #

shiftVals =: decr@(n base integers@(* insert))
shiftBy =: shiftVals@:(dim copies 3:)

NB. Updated to use shiftBy as function, not data
neighborArray =: shiftBy (shift"1 _) n
listNeighbors =: (+ p_insert)@:neighborArray - n

NB. Rules of the game, as an array
NB.      0 (cell dead)   1 (cell alive)
rules =: (3 = ])       ‘ ([: +./ 2 3 = ])

NB. Uses cell state to index into rule to apply
NB. appliedBy =: @.
nextState =: rules @. (cell =: [)("0)(p_rank 1) listNeighbors
```

The most significant change from the previous approach is changing `shiftBy` to a function that calculates the appropriate values based on the dimension of the board, rather than

```
   show board3D =: 2 2 2 reshape 0 0  0 1  1 0  1 1
0 0
0 1

1 0
1 1
   0 1 13 24 25 from shiftBy board3D
_1 _1 _1
_1 _1  0
 0  0  0
 1  1 _1
 1  1  0
   NB. Box places its arguments in a box
   NB. This displays each 3D array
   NB. the shift values from above
   box =: <
   box ("3) 0 1 13 24 25 from neighborArray board3D
+---+---+---+---+---+
|0 0|0 0|0 0|0 1|1 1|
|0 0|0 0|0 1|0 0|0 0|
|   |   |   |   |   |
|0 0|0 0|1 0|0 0|0 0|
|0 0|0 0|1 1|0 0|0 0|
+---+---+---+---+---+
```

Figure 4.3: Neighbors in "Game of Life", extended to any dimension

hard coding them like before. This change accomplished with the *base* function using much the same approach as found with the previous problem in Section 4.2.4, so the details of its behavior will be glossed with a simple illustration, found in Fig 4.3:

There is one additional change necessary: since the values to shift always include the "no shift" vector (a vector with only 0s as entries), without additional logic live cells will always be counted as having one additional neighbor. The listed code handles this by subtracting from the sum of the values the value of the cell itself. No other changes to the code are necessary for this extension to work, and thus the inherent parallelism of the solution for fixed rank is preserved without additional logic.

4.4 Merge Sort

4.4.1 Description

Conceptually, sorting an array using merge sort is fairly easy to describe.

- If the array has fewer than 2 elements, it's sorted
- Otherwise, divide the array into two halves and sort each of these
- Take the two sorted halves and merge them such that the resulting array remains sorted.

To simplify the discussion of the problem, we will be concerned with sorting arrays whose lengths are of the form 2^n, where n is a positive integer. This way the nature of the parallelism in the problem can be discussed without going into the detail of handling special cases.

Unfortunately, merge sort does not have a very intuitive extension into higher dimensions, since comparing collections is not similar to comparing numbers. However, it will be shown that an advantage of function rank in approaching this problem will be easily extending a given vector to sort into an arbitrarily high dimension.

4.4.2 Relevant Design Patterns

This problem represents the "Divide and Conquer" design pattern, in which, like its sequential equivalent, "the problem is solved by splitting it into a number of smaller sub-problems, solving them independently [concurrently], and merging the sub-solutions"[13]. Normally, problems that fit this design pattern are difficult to approach with data parallelism. Given the constraints mentioned above, however, merge sort will always the same number of sub-problems each with the same size, allowing for a data parallel approach.

4.4.3 Solutions

C with OpenMP

A parallel merge sort in C with OpenMP is given below. The solution, as well as the description of its workings, was taken from an article on parallelism in OpenMP and MPI[18].

```
void mergesort_parallel_omp
(int a[], int size, int temp[], int threads) {
    if ( threads == 1) {
        mergesort_serial(a, size, temp);
    }
    else if (threads > 1) {
        #pragma omp parallel sections
        {
            #pragma omp section
            mergesort_parallel_omp(a, size/2, temp, threads/2);
            #pragma omp section
            mergesort_parallel_omp(a + size/2, size-size/2,
                temp + size/2, threads-threads/2);
        }

        merge(a, size, temp);
        } // threads > 1
}
```

The `sections` compiler directive specifies that the following blocks of code (each beginning with a `section` directive) are to be executed independently of each other[16]. Then, after each sub-array is sorted, each thread (spawned by a different *section* directives) recursively merges (in-order) its sub-arrays in parallel, resulting in the original array being sorted.

While this solution is expressed very imperatively, most likely for performance reasons, it need not have been. Instead, each thread could have allocated memory and returned a new array with all the elements of the previous array in sort order. However, in either of these cases the programmer must explicitly work around the imperative nature of the programming language, either by manually requesting new memory safe for the stateful operations, or by manually keeping track of memory locations to avoid operating on data in use by other threads. However, this added effort would remain regardless of whether the solution were sequential or parallel, since it comes from the C programming language, and is typical of many imperative programming languages.

J

In the following J code, the function `merge` and its helper functions were omitted. The full solution can be found in Appendix D

```
p_rank =: "

NB. If the two items are in order, return them
NB. otherwise, reverse their order
inOrder_2 =: <:/
reverse =: |.
identity =: ]
sort2 =: reverse‘identity@.inOrder_2

NB. Turn rank1 array with 2^n elements
NB. Into rank n array, shape n copies of 2
NB. e.g., array of 8 => array of 2 2 2
divide =: $~ 2 #~ 2 ^. #

NB. Divide into different rank 1 cells (rows of 2 each)
NB. Sort each of these rank 1 cells
NB. Then, merge the rank 1 cells pairwise
NB. as many times as 1 less than the dimension of the rank
dim    =: #@$
repeated =: ^:
repeatedMerge =: merge/(p_rank 2) repeated (dim - 1:)
mergeSort =: repeatedMerge@:(sort2 p_rank 1)@:divide
```

This solution takes advantage of the constraints on the length of the array (it must be an integral power of two), expressing the regularity of the sizes of sub-problems through a regular array whose rank grows as the number of sub-problems does. For example, consider the code in Fig 4.4:

```
   NB. Deal: Create a list of integers
   NB. from the argument, and shuffle them
   deal =: ?~
   show vec8 =: deal 8
0 5 1 2 3 4 7 6
   show vec16 =: deal 16
11 8 3 10 12 1 15 9 4 13 7 14 6 5 2 0
   NB. Converts v8 into a 2 by 2 by 2 array
   divide vec8
0 5
1 2

3 4
7 6
   NB. Converts v16 into a 2 by 2 by 2 by 2 array
   divide vec16
11  8
 3 10

12  1
15  9


 4 13
 7 14

 6  5
 2  0
```

Figure 4.4: Divide and conquer by reshaping to an arbitrary rank array

Unlike the high level description of the problem given at the beginning of this section (but equivalently), this solution first sorts each two item vector, expressing the recursive base-case of the merge sort algorithm as a data parallel operation. Then, vectors are repeatedly merged via an insert of the `merge` function on the rank 2 items (which always consist of two vector elements). Again, while repeated function application doesn't quite fall into the bounds of data parallelism, each merge operation on the rank 2 items is itself data parallel, since the operation acts on the two-dimensional sub-collections concurrently. The example in Fig 4.5 illustrates this possibly unintuitive result of finding data parallelism in a traditionally divide and conquer problem.

```
   NB. Visualizing the application of the first
   NB. merge operation on vec16
   'merging' link ("2) sort2 ("1) divide vec16
+-------+----+
|merging|8 11|
|       |3 10|
+-------+----+
|merging|1 12|
|       |9 15|
+-------+----+

+-------+----+
|merging|4 13|
|       |7 14|
+-------+----+
|merging|5 6 |
|       |0 2 |
+-------+----+
   NB. Visualizing the application of the second
   NB. merge operation on vec16
   'merging' link ("2) (merge insert)(p_rank 2) sort2 ("1) divide vec16
+-------+---------+
|merging|3 8 10 11|
|       |1 9 12 15|
+-------+---------+
|merging|4 7 13 14|
|       |0 2  5  6|
+-------+---------+
```

Figure 4.5: Merging each of the vector elements of every matrix

Chapter 5

Results

5.1 Methodology

Each Parallel-J solution to the problems listed in Chapter 4 was timed running on varying problem sizes using 1, 2, and 4 threads For comparison, the performance of the solutions in C with OpenMP for the same problems is also listed. These C solutions were run on larger problem sizes in order to demonstrate their scaling more effectively (see Section 6.1 for more details). The Scala API Benchmark object was used to measure the performance of the Parallel-J solutions, which allowed for ease of use from the command line, automatic timing, and forcing the Java garbage collector to run between program runs. For the C programs, timing had to be done manually for each program. For all program results, the run time is given in seconds.

All solutions were run (and in the case of the C solutions, compiled) on a workstation with 16 1.0GHz *AuthenticAMD* processors each with 512 KB cache size. For the Parallel-J solutions (all of which can be found in Appendix C), a single trial consisted of running each program on a specific problem size with a specific number of threads a total of 6 times and measuring their execution time. the results of the latter 5 program runs were averaged. The first program run was omitted in order to account for the Java Just-In-Time (JIT) compiler making optimizations on the running Parallel-J code. For the solutions in C with OpenMP, a single trial consisted of running each program with a specific number of threads a total of 5 times. The performance for all the program runs for the C trials were averaged.

5.2 Calculating Pi using Numerical Integration

The number of operations performed in this problem scales as the number of divisions n between the interval $[0, 1]$ grows. For our tests, we measured performance for $n = 10,000$, $n = 20,000$, ..., $n = 100,000$. Table 5.1 below gives the outcomes.

	$n = 10,000$			$n = 20,000$			$n = 30,000$			$n = 40,000$		
Threads	1	2	4	1	2	4	1	2	4	1	2	4
Parallel-J	10.1	10.1	10.2	31.5	38.4	38.5	70.6	90.8	91.0	126.2	152.5	152.8

	$n = 50,000$			$n = 60,000$			$n = 70,000$		
Threads	1	2	4	1	2	4	1	2	4
Parallel-J	173.7	236.0	235.6	251.0	361.0	361.0	342.2	490.2	491.3

	$n = 80,000$			$n = 90,000$			$n = 100,000$		
Threads	1	2	4	1	2	4	1	2	4
Parallel-J	651.6	653.7	653.3	826.1	769.3	814.5	1017.5	1020.0	1021.5

	$x = 1,000,000$		
Threads	1	2	4
C with OpenMP	0.0233465	0.0117569	0.01764

Table 5.1: Performance results for calculating π

5.3 Game of Life

For the two-dimensional version of the "Game of Life", the number of operations performed in this problem scales as the dimensions x and y grow larger, and also as the number of desired iterations i grows. The latter, however, is fundamentally sequential, so i was fixed to be relatively small with value 10. In order to simplify the benchmarking process, trials were only run on square boards ($x = y$). Table 5.2 below gives the outcomes for x and y ranging from 100 to 300 by increments of 25.

	$x = 100$			$x = 125$			$x = 150$		
Threads	1	2	4	1	2	4	1	2	4
Parallel-J	66.4	66.5	66.4	159.8	159.6	159.9	326.8	335.5	348.1

	$x = 175$			$x = 200$			$x = 225$		
Threads	1	2	4	1	2	4	1	2	4
Parallel-J	725.8	728.3	726.1	1234.2	1231.2	1233.5	1979.8	1973.1	1973.3

	$x = 250$			$x = 275$			$x = 3$	
Threads	1	2	4	1	2	4	1	2
Parallel-J	2.45014e6	3.00412e6	2.62603e6	3.79871e6	3.79696e6	4.37458e6	5.36684e6	5.3685

	$x = 1000$		
Threads	1	2	4
OpenMP with C	1.22818	0.684292	0.683609

Table 5.2: Performance results for the Game of Life

5.4 Merge Sort

The number of operations required for this problem grows as the number n of values to sort increases. Because of the problem constraint listed in Section 4.4, problem sizes of the form 2^n were chosen, where $14 \leq n \leq 16$. The reader should note that due to time constraints, the Parallel-J version of this solution uses a `merge` function implemented imperatively in Scala, rather than implementing it in an equivalent way to the J idiom given in Appendix D. Table 5.3 below gives the outcomes.

	$n = 2^{14}$			$n = 2^{15}$			$n = 2^{16}$		
Threads	1	2	4	1	2	4	1	2	4
Parallel-J	2.9	2.5	2.5	10.6	9.2	9.2	41.1	35.5	35.7

	$x = 1000$		
Threads	1	2	4
OpenMP with C	0.413132	0.218665	0.222577

Table 5.3: Performance results for merge sort

Chapter 6

Conclusion

6.1 Discussion of Results

There are several potential reasons for the underwhelming results for the Parallel-J solutions given in Section 5. Probably the most significant reason for these results is that the body of code being parallelized (see Appendix B) potentially creates a large number of permanent and temporary objects, meaning that there are multiple requests for memory from the Java run-time heap. For example, since this work was originally intended to be a parallel implementation of J, several wrapper classes for were created to hold the same kind of information the current version of J[7] maintains about arrays and the primitive values, like reference counts and bit flags for type information, and so where in a normal Scala library there would have been operations on primitive values, in Parallel-J these operations were encumbered by having to work on instances of wrapper classes. The constant creation of new objects for every single integer likely lengthened program run-time significantly.

Also, operations that took existing collections and turned them into collections of higher rank were implemented by copying whole collections, operating on them, then concatenating each sub-result to return the final result. This constant copying of sub-arrays could have been avoided through the use of *index transformation* which have been described in Section 2.2.1, potentially improving performance significantly by removing these unnecessary operations.

Additionally, the large number of wrapper classes for primitive values increased each Parallel-J solution's memory requirements such that even comparatively modest problem sizes would not fit in cache, degrading performance significantly. This was first discovered when analyzing the performance results in Section 5.2, in which increasing the problem size by a factor of 10, from $10,000$ to $100,000$, caused the program's run time to increase by a factor of 100. The solutions in C with OpenMP, in contrast, did not display this behavior, and instead only began showing significant improvements from parallelism in the same problem at a problem sizes of $1,000,000$.

Finally, this research did not make use of the macros introduced by Scala 2.10[21] because Scala 2.10 was released too late in the research's development to be incorporated. These macros have the potential to improve performance by replacing at compile time source code expressions that appear to be acting on objects with lower-level expressions acting on primitives, thus avoiding new object creation.

6.2 Future Research

6.2.1 Parallel Implementation of J

This research would be useful for future work towards a parallel alternative of J. Regardless of the language chosen for implementing a parallel J, researchers must still consider which language features to include first in their prototype (Section 1.2) and must still understand the inherently data parallel nature of the rank operator (Section 2.1.4) Also, the forces that lead to the choice of Scala as the language for this research (also listed in Section 1.2), should suggest to future researchers the kinds of language features desirable for implementing a parallel J. If future work is done in this area in Scala or another language with suitable programming paradigms and libraries for developing a parallel implementation of J, then this research should be used as a prototype to help guide development.

Alternatively, a future parallel implementation of J could be done in the C programming language, based on the current implementation of J[7]. In order to begin this approach, it is strongly recommended that researchers first familiarize themselves with Roger Hui's documentation, "An Implementation of J." Both this documentation and the full source code (under open sources licenses) are available freely on the J Programming web site[9]. In addition, it seems that there are at least two viable options in such an approach which are possibly not mutually exclusive. One option is to use a shared memory parallel environment such OpenMP to parallelize operations within a single instance of a parallel J. Another is to take advantage of the fact that the current implementation of J already includes functionality to allow for multiple instances of J to be running in the same process without race conditions and approach future work using a distributed memory parallel environment, such as MPI[3].

6.2.2 Sequential and Parallel Scala Libraries for Regular Arrays using Function Rank

Another possible extension to the work presented here is the development of a Scala library for arbitrary dimensional collections that uses function rank. This library would ideally support parallelism in much the same way Scala's current collections library does[17], through conversions between sequential and parallel implementations of each collection type. However, it should be clear that even a purely sequential version of such a library would be useful for solving problems which requires operations on several different dimensions, or which are naturally expressed in a higher dimension than the original problem description.

At the time this research was conducted, the author was not aware of current work being done in generic and polytipic programming in Scala by Miles Sabin. In particular this work, which is collectively called "Shapeless"[19], supports collections whose sizes are known statically. This functionality would be useful for future work in developing a library of collections whose dimensions are known statically, possibly granting some or all of the advantages give when discussing the Haskell library *Repa* in Section 2.2.1[12]. Combining these advantages with functions supporting function rank into a single Scala library would lead to significant expressiveness and reduced boiler-plate code, and could lend itself to a future parallel implementation for good performance, as well.

6.2.3 Implications for Other Work in Data Parallelism

This paper demonstrates that, conceptually, the formalization of function rank found in the functional and collections oriented J programming language and Parallel-J Scala library can help programmers exploit potential concurrency in the form of data parallelism, because it allows programmers a take existing functions and, in an inherently data parallel fashion, both apply them to specific ranks of a collection and extend them to similar problems in higher dimensions. There may be other, equivalent models of function rank expressed in other programming paradigms, such as object oriented programming, used in Parallel-J to express the rank operator; developments in type theory may lead to elegant static expressions of the shape of regular collections and a function's shape (instead of just rank) requirements and transformations of its data. Whatever the future holds in the field of parallel computing, this paper hopes to make researchers aware that an elegant (and race-condition safe) solution already exists to an entire class of data parallel problems and is waiting for its time to be acknowledged as, and used in conjunction with, other great ideas in the field.

Bibliography

[1] Dyalog Limited. Dyalog apl: Version 13.2 documentation centre, 2012.

[2] Ronald Garcia et al. The Boost Multidimensional Array Library. Boost C++ Libraries `http://www.boost.org/doc/libs/1_53_0/libs/multi_array/doc/user.html`.

[3] The MPI Forum. MPI: A message passing interface, 1993.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Martin Gardner. Mathematical Games - The Fantastic combinations of John Conway's New Solitaire Game "Life". Scientific American 223. pg 120-123, 10 1970.

[6] Roger K. W. Hui and K. E. Iverson. J introduction & dictionary. J Software, Inc. `http://www.jsoftware.com/help/dictionary/vocabul.htm`.

[7] Roger KW Hui. An Implementation of J. Iverson Software, Inc, 1992.

[8] Roger KW Hui. Rank and Uniformity. In *APL '95 Proceedings of the international conference on Applied programming languages*. ACM SIGAPL, 1995.

[9] Jsoftware Inc. Jsoftware: High-performance development software.

[10] K. E. Iverson. Operators and Functions. Technical report, IBM Thomas J. Watson Research Center, 1978.

[11] K. E. Iverson. Rationalized APL. Technical report, I.P. Sharp Associates, 1983.

[12] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *ACM SIGPLAN Notices*, pages 261,272, 2010.

[13] Timothy G. Mattson, Beveryl A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming.* Addison-Wesley, 2010.

[14] Martin Odersky. Scala 2.8 Collections, 2009.

[15] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the scala programming language. Technical report lamp-report-2006-001, EFPL, 2006.

[16] OpenMP Architecture Review Boad. OpenMP application program interface. `http://www.openmp.org`.

[17] A. Prokopec, P. Bagwell, T. Rompf, and M Odersky. A generic parallel collection framework. *Euro-Par 2011 Parallel Processing*, pages 136–147, 2011.

[18] Atanas Radenski. Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps. *Proc. PDPAT*, 11:367–373.

[19] Miles Sabin. Shapeless: An exploration of generic/polytypic programming in scala. `https://github.com/milessabin/shapeless`.

[20] S. B. Scholz. Single-Assignment C - efficient support for high level array operations in a functional setting. *Journal of Functional Programming*, 2003.

[21] École Polytechnique Fédérale de Lausanne (EPFL). Scala 2.10, 2013.

Appendix A

A Partial J Lexer Written in Scala

```
package j.lang

import scala.annotation.tailrec

object JLexer {

        class JLexeme(val chars: String, val state: JState.State) {
          def +:(str: String) = JLexeme(this.chars + str, state)
          override def toString = "JLexeme("+chars+","+state+")"
        }

        object JLexeme {
          def apply(chars: String, state: JState.State) =
                new JLexeme(chars, state)
          def apply(chars: Seq[Char], state: JState.State) =
                new JLexeme(chars.mkString(""), state)
        }

        object JState extends Enumeration {
          type State = Value
          val Space             = Value(0)
          val Other             = Value(1)
          val AlphNum           = Value(2)
          val N                 = Value(3)
          val NB                = Value(4)
          val `NB.`             = Value(5)
          val Numeric           = Value(6)
          val Quote             = Value(7)
          val EvenQuotes        = Value(8)
          val Comment           = Value(9)
        }

        object JCharClass extends Enumeration {
          type CharClass = Value
          val Other             = Value(0)
          val Space             = Value(1)
          val AlphNotNB         = Value(2)
          val N                 = Value(3)
          val B                 = Value(4)
          val Numeric           = Value(5)
          val Period            = Value(6)
          val Colon             = Value(7)
          val Quote             = Value(8)
        }

        import JState._

        object SMFuncCode extends Enumeration {
          type FuncCode = Value
          val Pass              = Value(0)
          val NextWord  = Value(1)
          val EmitWord  = Value(2)
          val EmitWErr  = Value(3)
          val EmitVect  = Value(4)
          val EmitVErr  = Value(5)
          val Stop              = Value(6)
        }

        import SMFuncCode._
        object SMFuncRes {
          def apply(s: Int, c: Int) = new SMFuncRes(JState(s),SMFuncCode(c))
        }

        class SMFuncRes(val state: JState.State, val code: FuncCode) {
          override def toString = "(" + state + "," + code + ")"
        }

// Taken from the J Vocabulary page for Sequential Machine
        val smLookUpTable = """
' X    S    A    N    B    9    D    C    Q ']0
 1 1  0 0  2 1  3 1  2 1  6 1  1 1  1 1  7 1  NB. 0 space
 1 2  0 3  2 2  3 2  2 2  6 2  1 0  1 0  7 2  NB. 1 other
 1 2  0 3  2 0  2 0  2 0  2 0  1 0  1 0  7 2  NB. 2 alp/num
 1 2  0 3  2 0  2 0  4 0  2 0  1 0  1 0  7 2  NB. 3 N
 1 2  0 3  2 0  2 0  2 0  2 0  5 0  1 0  7 2  NB. 4 NB
 9 0  9 0  9 0  9 0  9 0  9 0  1 0  1 0  9 0  NB. 5 NB.
 1 4  0 5  6 0  6 0  6 0  6 0  6 0  1 0  7 4  NB. 6 num
 7 0  7 0  7 0  7 0  7 0  7 0  7 0  7 0  8 0  NB. 7 '
 1 2  0 3  2 2  3 2  2 2  6 2  1 2  1 2  7 0  NB. 8 ''
 9 0  9 0  9 0  9 0  9 0  9 0  9 0  9 0  9 0  NB. 9 comment
""".split("\n").drop(2).map(_.drop(1).split("  ").dropRight(1).map(entry =>{
  val Array(s,c) = entry.split(" ").map(_ toInt)
  SMFuncRes(s,c)
}))

        import JCharClass._

        class CharWClass(val char: Char, val charclass: CharClass) {
          override def toString() = "(" + char + "," + charclass + ")"
        }

        object CharWClass {
                def apply(char: Char, charClass: CharClass) =
                        new CharWClass(char,charClass)

                def apply(char: Char) = new CharWClass(char, charClassify(char))

                val charClasses = (0 until 256).map((i:Int) =>
                        initCharClassify(i.toChar)).toArray

                def initCharClassify(c: Char)= {
                        import JCharClass._

                        if              (c == ' ')      Space
                        else if         (c.isDigit
                                || c == '_')            Numeric
                        else if         (c == 'N')      N
                        else if         (c == 'B')      B
                        else if         (c.isLetter)    AlphNotNB
                        else if         (c == '.')      Period
                        else if         (c == ':')      Colon
                        else if         (c == '\'')     Quote
                        else                            Other
                }

                def charClassify(c: Char) = charClasses(c.toInt)
```

```
}

private object SMRunningState {
  case class EmitState(r: State, k: Int)

  def apply(i: Int, j: Int, state: State) = {
          new SMRunningState(i,j,state, List(), None)
  }
}

private class SMRunningState private(
        private var ip: Int, private var jp: Int,
    private var sp: State, private var ap: List[JLexeme],
    private var vp: Option[SMRunningState.EmitState]) {

  def i = ip
  def j = jp
  def accum = ap
  def state = sp
  def evState = vp

  private def ev(line:Seq[CharWClass]) = {
    import SMRunningState._
            evState match {
              case None =>
                        JLexeme(line.slice(j, i).map(_ char),
                                        state) +: accum
              case v: Some[EmitState] => {
                if (v.get.r == state) {
                  accum.head.+:(line.slice(v.get.k,i).map(_
                  char).mkString) +: accum.drop(1)
                }
                else {
                  JLexeme(line.slice(j, i).map(_ char),state) +: accum
                }
              }
            }
  }

  def ew(line:Seq[CharWClass]) =
        JLexeme(line.slice(j, i).map(_ char),state) +: accum

  import SMRunningState._
  def next(fr: SMFuncRes, line:Seq[CharWClass]) = {
        ap = fr.code match {
          case Pass     => accum
          case NextWord => accum

          case EmitWord => ew(line)
          case EmitWErr => ew(line)

          case EmitVect => ev(line)
          case EmitVErr => ev(line)
        }
        vp = fr.code match {
          case Pass     => evState
          case NextWord => evState

          case EmitWord => None
          case EmitWErr => None

          case _ => Some(EmitState(state,i))
        }

        sp = fr.state
        jp = fr.code match {
          case EmitWErr => -1
          case EmitVErr => -1

          case Pass     => j

          case _        => i
        }
        ip += 1
  }

  private val endState = smLookUpTable(JCharClass.Space.id)
  def finalize(fr: SMFuncRes, line:Seq[CharWClass]) = {
          ap = ev(line)
  }
}

def sequentialMachine(line: String): List[JLexeme] =
  sequentialMachine(SMRunningState(0,-1,JState.Space),
        line.map(CharWClass(_)))

@tailrec def sequentialMachine(runState: SMRunningState,
    line:Seq[CharWClass]): List[JLexeme] = {
  import runState._
```

```
          if (line isEmpty)
            accum.reverse
          else if (i >= line.length) {
            val funcRes = smLookUpTable(state.id)(JCharClass.Space.id)
            runState.finalize(funcRes,line)
            accum.reverse
          }
          else {
            val funcRes = smLookUpTable(state.id)(line(i).charclass.id)
            runState.next(funcRes, line)
            sequentialMachine(runState,line)
          }
        }
}
```

Appendix B

An Implementation of Function Rank in Scala

```
package j.lang.datatypes.function

/*Imports*/

abstract class JVerb[M <: JArrayType : Manifest,
  D1 <: JArrayType : Manifest, D2 <: JArrayType : Manifest,
  MR <: JArrayType : Manifest, DR <: JArrayType : Manifest]
  (rep: String, val ranks: List[JFuncRank],
  mdomain: JTypeMacro,
  d1domain: JTypeMacro, d2domain: JTypeMacro) extends
  JFunc[JArray[M], JArray[D1], JArray[D2],
        JArray[MR], JArray[DR]](rep, jVERB,
                mdomain, d1domain, d2domain) {

  def apply[T <: JArray[M]](y: T) = monad(y)

  override def monad[T <: JArray[M]](y: T) = {
          val jaf = JArrayFrame(ranks.map(_ r1), y)

           val newCells = if (!JVerb.parallelFlag) {
            (0 until jaf.frameSize) map { fr =>
             monadImpl(JArray(jaf.jar.jaType, jaf.cellShape,
                        jaf.jar.ravel.slice(fr*jaf.cellSize,
                                (1+fr)*jaf.cellSize)))
          }}
           else {
            (0 until jaf.frameSize).par map { fr =>
            monadImpl(JArray(jaf.jar.jaType, jaf.cellShape,
                        jaf.jar.ravel.slice(fr*jaf.cellSize,
                                (1+fr)*jaf.cellSize)))
           }}

          val newShape = jaf.frames.dropRight(ranks.length
                ).foldLeft(List[Int]())(_ ++ _) ++ newCells(0).shape
          JArray(newCells(0).jaType, newShape,
                newCells.foldLeft(Vector[MR]())(_ ++ _.ravel))
        }

        def apply[T1 <: JArray[D1], T2 <: JArray[D2]](x: T1, y: T2) = dyad(x,y)

        override def dyad[T1 <: JArray[D1], T2 <: JArray[D2]](x: T1, y: T2) = {
          val jafx = JArrayFrame(ranks.map(_ r2), x)
          val jafy = JArrayFrame(ranks.map(_ r3), y)

          jafx.shapeAgreement(jafy) match {
            case None => throw new Exception()
            case Some(agree) => {
              val xreframed = jafx.shapeToNewFrame(agree)
              val yreframed = jafy.shapeToNewFrame(agree)

              val xcellShape = jafx.frames.last
              val xcellSize = xcellShape.foldLeft(1)(_ * _)
              val ycellShape = jafy.frames.last
              val ycellSize  = ycellShape.foldLeft(1)(_ * _)
              val frameSize  = agree.init.foldLeft(1)(_ * _.foldLeft(1)(_ * _))

              val newCells = if (!JVerb.parallelFlag) {
                (0 until frameSize) map { fr =>
                dyadImpl(
                                JArray(jafx.jar.jaType, xcellShape,
                                        xreframed.ravel.slice(fr*xcellSize,
                                                (1+fr)*xcellSize)),
                         JArray(jafy.jar.jaType, ycellShape,
                                        yreframed.ravel.slice(fr*ycellSize,
                                                (1+fr)*ycellSize)) )
              }}
              else {
```

```
                (0 until frameSize).par map { fr =>
                dyadImpl(
                                JArray(jafx.jar.jaType, xcellShape,
                                        xreframed.ravel.slice(fr*xcellSize,
                                                (1+fr)*xcellSize)),
                        JArray(jafy.jar.jaType, ycellShape,
                                        yreframed.ravel.slice(fr*ycellSize,
                                                (1+fr)*ycellSize)) )
              }}

              val newShape = agree.dropRight(1).foldLeft(
                        List[Int]())(_ ++ _) ++ newCells(0).shape
              JArray(newCells(0).jaType, newShape,
                        newCells.foldLeft(Vector[DR]())(_ ++ _.ravel))
            }
          }
        }

 def addRanks(r: JFuncRank) = {
    val outerRef = this
    new JVerb[M,D1,D2,MR,DR](
        rep + "(\"" + r + ")",
        ranks :+ r,
        mdomain, d1domain, d2domain) {

      override def monadImpl[T <: M : Manifest](y: JArray[T]) = {
        outerRef(y)
      }
      override def dyadImpl[T1 <: D1 : Manifest,
                T2 <: D2 : Manifest](x: JArray[T1], y: JArray[T2]) =
        outerRef(x, y)
    }
  }

        protected def monadImpl[T <: M : Manifest](y: JArray[T]): JArray[MR]
        protected def dyadImpl[T1 <: D1 : Manifest,
                T2 <: D2 : Manifest](x: JArray[T1], y: JArray[T2]): JArray[DR]
}

object JVerb {
  var parallelFlag = false
}
```

Appendix C

Using the Parallel-J System as a Scala Library

Below are the solutions to the problems listed in Chapter 4 using the Parallel-J system as a Scala library.

C.1 Calculating Pi with Numerical Integration

```
package j.test.benchmark.NumInt

/*Imports*/

abstract class NumIntBench extends Benchmark {

    var numSquares: JInt = null
    private var pi: JArray[JNumber] = null

    override def setUp()

    //modified from scala.testing.Benchmark
    override def main(args: Array[String]) {
        //command line argument parsing, setting value for numSquares
    }

    def run() {
      val recip = JArray.scalar(numSquares.recip)
      val xvals =  signumMultiply(integersIndex(JArray.scalar(numSquares)),
                              recip)
      val yvals = recipricalDivide(
          conjugatePlus(JArray.scalar(JReal.One),
              squareNotand(xvals)))

      pi = (conjugatePlus insert).apply(signumMultiply(
          yvals,
          signumMultiply(
              JArray.scalar[JInt,Int](4),
              recip)
          ))
    }

    override def tearDown() {
      println("Pi is " + pi)

      super.tearDown()
    }
}
```

C.2 Game of Life

```
package j.test.benchmark.GameOfLife

/*Imports*/

abstract class GOLBench extends Benchmark {
    var boardShape: JArray[JInt] = null
    val steps = 10
    val ratioAliveDead = new JFloat(0.5)

    //helper verbs
    val equals3 = new JVerb1Type[JInt](
        "(3 = ])",
        List(JFuncRank(0)),
        jINT){
        override def monadImpl[T <: JArrayType : Manifest](
            y: JArray[T]) =
                throw new NotImplementedException()

        override def dyadImpl[T1 <: JArrayType : Manifest,
            T2 <: JArrayType : Manifest](
            x: JArray[T1], y: JArray[T2]) = {
                equal(JArray.scalar[JInt,Int](3),
                      y)
            }
          }
    val equals2_or_3 = new JVerb1Type[JInt](
          "([: +./ 2 3 = ])",
          List(JFuncRank(0)),
          jINT){
            override def monadImpl[T <: JInt : Manifest](
                y: JArray[T]) =
                    throw new NotImplementedException()

            override def dyadImpl[T1 <: JInt : Manifest,
                T2 <: JArrayType : Manifest](
                x: JArray[T1], y: JArray[T2]) = {
                    (realOr insert).apply(equal(
                              JArray.vec2(2,3),
                               y)).asInstanceOf[JArray[JInt]]
            }
          }

    override def main(args: Array[String]) {
        //command line argumen parsing, setting value for boardShape
    }

    override def setUp()

    def run() {

      val numCells = (signumMultiply insert).monad(
            boardShape).asInstanceOf[JArray[JInt]]

      val lifeThreshold =
        signumMultiply(
            JArray.scalar(ratioAliveDead),
            numCells).asInstanceOf[JArray[JReal]]

      val board = shapeReshape(
          boardShape,
          incrementGreaterthanequal(
              lifeThreshold,
              rollDeal[JArray[JInt],JArray[JInt]](
                  numCells,
                  numCells))).asInstanceOf[JArray[JInt]]

      val shiftBy = shapeReshape(
          JArray.vec2(8, 2),//    DR      D      DL
          JArray.auto[JInt, Int](
          //DR      D      DL
           -1,-1,  -1,0,  -1,1,

          //R      L     UR     U     UL
            0,-1,  0,1,  1,-1,  1,0,  1,1)
          ).asInstanceOf[JArray[JInt]]

      val neighborArray = (y: JArray[JInt]) =>
            reverseShift(shiftBy, y).asInstanceOf[JArray[JInt]]

      val listNeighbors = (y: JArray[JInt]) =>
            (conjugatePlus insert).monad(neighborArray(y)
            ).asInstanceOf[JArray[JInt]]

      val nextState = leftIdentity.asInstanceOf[
            JVerb[JInt, JInt, JInt, JInt, JInt]] agenda(
          equals3, equals2_or_3)

      var boardvar = board

      for (i <- 0 until steps) {
        boardvar =
            nextState(boardvar, listNeighbors(boardvar)
            ).asInstanceOf[JArray[JInt]]
      }
    }
}
```

C.3 Merge Sort

```
package j.test.benchmark.MergeSort

/*Imports*/
```

```
abstract class MergeSortBench extends Benchmark {

    var y: JArray[JInt] = null
    var res: JArray[JInt] = null

     override def main(args: Array[String]) {
        //command line argument parsing, setting value for y
    }

    override def setUp()

    def run() {
      val jtwo = JArray.scalar(JInt(2))
      val j16  = JArray.scalar(JInt(16))

      val intReverse = reverseShift.asInstanceOf[
        JVerb[JInt, JInt, JInt, JInt, JInt]]

      val sort2 = (decrementLesserthanequal insert) agenda(
          reverse,
          rightIdentity.asInstanceOf[JVerb1Type[JInt]])

      val divide = (y: JArray[JInt]) => {
        shapeReshape(
            tallyCopies(
                naturalLog(
                    jtwo,
                    tallyCopies(y)).asInstanceOf[JArray[JFloat]].toJInt,
                jtwo).asInstanceOf[JArray[JInt]],
            y).asInstanceOf[JArray[JInt]]
      }

      val dim = (y: JArray[JInt]) => {
        tallyCopies(shapeReshape(y))
      }

      val sortBase = (sort2 addRanks(JFuncRank(1)) )
      val merger2 = (merge insert) addRanks(JFuncRank(2))

      val repeatedMerge = (y: JArray[JInt]) => {
        merger2.power(decrementLesserthanequal(dim(y)
            ).asInstanceOf[JArray[JInt]]).apply(y)
      }
      def mergeSort = (y: JArray[JInt]) => {
        repeatedMerge(sortBase(divide(y)))
      }

      res = mergeSort(y)
    }

    override def tearDown() {
      println("Array is sorted: " +
          (decrementLesserthanequal insert).apply(res) )
      super.tearDown()
    }
}
```

Appendix D

Merge Sort in J

For the sake of simplicity some of the functions used in the J implementation of merge sort lised in 4.4.3 were elided. The full program is given below.

```
NB. ‘mrg’ Taken from the J Phrase Book
mrg =: 1 : ’/:@/:@(m" _) { ,’

merge =: 4 : 0
b =. x interleaveOrdered y
y (b mrg) x
)
interleaveOrdered =: i.@:+&# e. (+ i.@:#)@:(+/"1@:>("0 1))

sort2 =: |.‘]@.(<:/)

sort2 =: |.‘]@.(<:/)
divide =: $˜ 2 #˜ 2 ^. #
dim    =: #@$
mergeSort =: (merge/("2) ^: (dim - 1:))@:(sort2"1)@:divide
```

Appendix E

J #: (Base) Primitive

The J primitive `#:` is a function that allows programmers to easily convert numeric values into different number bases. For its one argument (*monadic*) usage, the primitive defaults to converting values into base 2. The result of applying this function as a monad is a vector of 0s and 1s, with the indicies giving the appropriate place value.

```
   base =: #:
   base 2
1 0
   base 3
1 1
   base 65
1 0 0 0 0 0 1
```

For its two argument (*dyadic*) use case, `#:` takes as its left argument a vector representing the place values of the desired base, shown below.

```
   2 2 base 3
1 1
   2 2 2 2 2 2 2 base 65
1 0 0 0 0 0 1
   10 10 base 65
6 5
```

One advantage of taking a vector for a numeric base representation, instead of a more traditional scalar (base 2, 8, 10, 16, etc.), is that that `#:` can also represent numbers in irregular numeric bases. The example below shows how `#:` can be used to represent 100,000,000 milliseconds in the most familiar of the irregular numeric bases, time as measured in days, hours, seconds, and milliseconds.

```
   365 24 60 60 1000 base 100000000
1 3 46 40 0
```

In the discussion of extending the Game of Life to higher dimensions in Section 4.2.4, the dyadic case of `#:` is used in the function `xs`, which takes a vector `vn` of length `n` specifying how to divide the interval $[0, 1]^n$. `#:`'s purpose in `xs` is to convert scalar values to a vector `vcn` representing coordinate values, where each scalar `c` at index `i` in `vcn` lies in the interval $[0,$`i from vn`$]$. Finally, each vector element in `vcn` is divided by `vn` so that the resulting coordinate values all lie in the interval $[0, 1]^n$. This process is illustrated step by step in the following example.

```
   intergers =: i.
   insert =: /
   (* insert) 4 4
16
   integers (* insert) 4 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   4 4 base integers (* insert) 4 4
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
2 0
2 1
2 2
2 3
```

```
3 0
3 1
3 2
3 3
   (4 4 base integers (* insert) 4 4) %(" 1) 4 4
   0    0
   0 0.25
   0  0.5
   0 0.75
0.25    0
0.25 0.25
0.25  0.5
0.25 0.75
 0.5    0
 0.5 0.25
 0.5  0.5
 0.5 0.75
0.75    0
0.75 0.25
0.75  0.5
0.75 0.75
```

Glossary

agreement (Spelled: `; "`) Higher order function that takes a function and returns a function that visualizes the shape agreement of its arguments under the argument function. 12

base (Spelled: `#:`) When given one argument, converts a number into binary, with output an array of bits. When given two arguments, left argument is a vector used as the base (radix) for representing the right argument; output is again an array. 38

deal (Spelled: `?~`) Function that creates a list of numbers like when "integers" is given a scalar argument, only randomly shuffled. 49

foreign (Spelled: `!:`) Higher order function taking two arguments that index, from left to right, a library and a function.. 24

from (Spelled: {) Function that takes from the right argument the items specified by the right argument. 11

increment (Spelled: `>:`) Function that takes an array of numbers and increments them (adds 1 to each number). 26

insert (Spelled: `/`) Higher order function that takes a function and inserts it between the items of an array, starting from the right. Equivalent to reduceRight or foldRight in Scala. 28

integers (Spelled: `i.`)Function that creates an array of integers with the shape of the argument array. Populated by incrementing values, starting at 0. 11

NB. Nota bene. Begins a comment spanning a line. 11

reshape (Spelled: `$`) Function that creates an array by associating the left argument as the shape of the right argument. 43

shape (Spelled: `$`) Function that takes one argument and returns a vector specifying its shape. 39

show (Spelled: `]`) Identity function for the right (as opposed to left) argument. Used to display the a value that has been assigned a name, which by default would not display. 11

showTasks (Spelled: `(; ") (< @)`) Higher order function that takes a function and returns a function that visualizes the parallelizable tasks that would be created from the parallel rank operator. 26