Spartns

Jerônimo Pellegrini

February 28, 2009

1 Introduction

Although Common Lisp is usually considered a big language, it has no built-in support for sparse matrices or tensors except for hashtables, which can be slow and awkward.

Spartns is a sparse array representation library for ANSI Common Lisp. It has been built to be very flexible while also being very efficient.

Spartns allows for different dimensions of an array to be represented in different ways:

- *Hashtables*: although not efficient, this representation scheme was included to illustrate the library's flexibility. Given an index specification $[ijk]$ and en element e, we can store the element in a hashtable h indexed by the list `'(i j k)` by doing `(setf (gethash (list i j k) h) e)` (Spartns will actually use three hashtables in this case:
`(setf (gethash k (gethash j (gethash i h))) e)`);

- *Compressed vectors*: if we know that the sparsity structure will not change for certain array dimensions, we can represent them as compressed vectors. A compressed vector is a pair of arrays, one for the indices and one for the values. To access an element, we first do a binary search on the indices array, and then access the element in the values array;

- *Plain arrays*: if we know that a specific dimension of the tensor is dense, we can represent that dimension as an array.

Spartns is both flexible and efficient because it was designed to produce macros and functions with type declarations for each kind of tensor you specify. For example, if a tensor has rank euqal to four, the number of non-zero elements is one hundred in each dimension and you want the three first dimensions to be represented as hashtables, but the last one as compressed vectors, the element type is double-float, and the sparse element is -1, you can define a spartn scheme like this:

- representation '(hash hash hash cvector)

- non-zeros '(100 100 100 100)
- element-type double-float
- sparse-element -1d0

2 Simple usage instructions

The macro `defspartn` creates a new tensor representation type and defines functions to get, set and delete elements as well as traverse the structure and check the number of non-zero elements. The library uses the name *"spartns"* for sparse tensor types.

The first argument to `defspartn` is a name to the structure. You can, for example, define a structure where the first dimension is represented as a hashtable and the second as compressed vectors, and the element type is double-float – and call it "HCD":

```
(defspartn hcd
    :representation (spartns::hash spartns::cvector)
    :non-zero       (3 4)
    :element-type   double-float
    :sparse-element 0d0)
    :test-equal     eql
```

The name of the structure will be used to create the functions:

- `get-hcd`
- `set-hcd`
- `delete-hcd`
- `copy-hcd`
- `pack-hcd`
- `traverse-hcd`

The function used to test element equality is `EQL`, but any other could have been used.

Each time a container is created to map one index into the next dimension, it is created with the size specified by the `non-zero` parameter. So, for the example above the `(3 4)` argument to `non-zero` means that:

- The first dimension is a hashtable that is created as `(MAKE-HASH-TABLE :SIZE 3)`;
- For each index in the first dimension that is set, a cvector is created with size 4.

```
(let ((mat (make-hcd)))
  (set-hcd 1 2 5d0)
  (get-hcd 1 2)     ; returns 5d0
  (get-hcd 1 1)     ; returns 0d0, the sparse element
  (traverse-hcd ((i j) val mat)
    (format t "mat[~a ~a]=~a~%" i j value)))
```

The `traverse-hcd` macro was created to traverse `hcd` structures. The symbols i and j will be bound to dimension indices; val will be bound to the value stored at (i,j), and mat is the HCD-structure.

Besides those, there is also the macro `spartn-convert`, which converts between structures:

```
(defspartn ccd
    :representation (spartns::cvector spartns::cvector)
    :non-zero       (3 4)
    :element-type   double-float
    :sparse-element 0d0
    :def            T)

(let ((mat (make-hcd)))
  (set-hcd 1 2 5d0)
  (set-hcd 1 1 2d0)
  (let ((new-mat (spartn-convert mat
                                 :from hcd
                                 :to ccd
                                 :destructuve t)))
    (traverse-ccd ((i j) value new-mat)
      (format t "new-mat[~a ~a]=~a~%" i j value))))
```

The `DEF` parameter for `defspartn` means that you want it to immediately create the functions and macros that you will use to access the tensors.

The macro `w/spartn` creates local functions (with `LABELS` instead of `DEFUN`), and will allow you to include aliases for spartn schemes:

```
(w/spartn (cchd)
  (let ((m make-cchd))
  (set-cchd i j k 10.5)
  ...)
```

If you use `w/spartns` you don't need to set `DEF` to `T` in `defspartn`, because `w/spartns` will create local functions and macros with `LABELS` and `MACROLET`.

2.1 Traversals

When `defspartns` is used to create a new structure, a macro is also created to traverse it.

```
(defspartn 2dmat
  :representation (cvector array)
  :element-type   symbol
  :sparse-element 'NOTHING
  :def            T)
```

Any object created with `make-2dmat` can be traversed with `traverse-2dmat`:

```
(let ((mat (make-2dmat)))
  (set-2dmat mat 0 3 'X)
  (set-2dmat mat 1 2 'Y)
  (traverse-2dmat ((a b) val mat)
      (format t "mat[~a ~a] = ~a~%" a b val)))
```

The output from the code above is

```
mat[0 3] = X
mat[1 2] = Y
```

2.1.1 Partial traversals (keeping some indices fixed)

When traversing, it is possible to keep some indices fixed while traversing over the others. For example, this

```
(let ((j 5))
  (traverse-3dmatrix ((i j k) v mat :dont-bind j)
    (format t "mat[~a ~a ~a] = a~%" i j k v)))
```

will show all values of $mat[ijk]$ for all i and k, `with` $j = 5$.

2.2 Traversing multiple tensors

You may want to traverse several tensors at once. For example, matrix multiplication can be described as this:

```
(w/spartn-traversals ((ccd X m n val-x)
                      (ccd Y n o val-y))
  (set-ccd Z m o (+ (get-ccd Z m o)
                    (* val-x val-y))))
```

That will be expanded into:

```
(TRAVERSE-CCD ((M N) VAL-X X)
  (TRAVERSE-CCD ((#:Y-N-3942 K) VAL-Y Y)
    (WHEN (AND (EQL N #:Y-N-3942))
      (PROGN
        (SET-CCD* Z M O
          (+ (GET-CCD* Z M O)
             (* VAL-X VAL-Y))))))
```

Note that there will be an implicit `PROGN` around the traversal body.

It is clear from the expanded code above that thisis not best way to do the traversals: the code runs through all elements of all tensors, even when indices don't match, and uses `WHEN` to skip the body. Spartns currently does not support faster traversals, but this is planned for future releases.

2.3 SETF-able values when traversing

During traversals you can `SETF` the value:

```
(w/spartn-traversals ((ccd X m n val-x)
                      (ccd Y n o val-y))
  (setf val-y (some-function)))
```

2.4 Traversing multiple tensors, repeatedly

If you know that the sparsity structure of your tensors will not change and you need to traverse them several times, you may try the macro `w/fast-traversals`. It will call `w/spartn-traversals` once, collecting only the elements that are non-zero for all index combinations into arrays, and later traverse only those arrays. The syntax is the same as for `w/spartn-traversals`:

```
(w/fast-traversals ((ccd X m n val-x)
                    (ccd Y n o val-y))
  (while (not (converged))
    ...))
```

Fast traversals are different from normal traversals in some aspects:

- There is a setup overhead, while the sparsity structure is learned (all tensors are traversed once). After that, only their intersection will be traversed;
- Values are not `SETF`-able during traversals.

2.5 From a dynamic representation to a static one

If you want to include an unknown number of elements in a tensor, you can:

- Add elements to a tensor represented as `CVECTOR`. In this case, you should add indices in order, or this process may take a long time (adding indices out of order makes Spartns re-arrange the internal vector repeatedly to keep indices sorted);
- Use the `HASH` scheme. It may or may not be fast, depending on your Lisp implementation;
- Add elements to a tensor represented as `HASH`, and later convert to `CVECTOR` for fast traversals. This second approach is described in the rest of this subsection.

You can create the static spartn type with an approximate number for `:non-zero`:

```
(defspartn static-2d
  :representation (cvector cvector)
  :non-zero       (200 200)
  :resize-amount  30
  :sparse-element 0d0
  :element-type   double-float)
```

The `resize-amount` parameter is the amount by which compressed vectors will grow after they run out of space. If you set it too high, you may need more memory. If you set it too low, Spartns will keep resizing vectors too often. Another way to specify how each cvector grows is by using the `resize-function`:

```
(defspartn static-2d
  :representation  (cvector cvector)
  :non-zero        (200 200)
  :resize-function (lambda (n) (* 2 n))
  :sparse-element  0d0
  :element-type    double-float)
```

The example above will double the size of a cvector when it is full.

Then you can convert from dynamic to static and pack the static representation, so the extra memory can be freed:

```
(let ((static-mat (spartn-convert dyn-mat
                                  :from dyn-2d
                                  :to   static-2d
                                  :destructive T)))
  (setq static-mat (pack-static-2d MAT))
  ...)
```

3 Creating new representation schemes

Besides the two representation schemes that come out-of-the-box with Spartns (`cvector` and `hash`), you can add any other by using the `defscheme` macro. This is how the `hash` scheme was added:

```
(defscheme :name     hash
           :type     hash-table
           :make     (make-hash-table :size size)
           :get      (gethash index data sparse-element)
           :set      (setf (gethash index data) value)
           :delete   (remhash index data)
           :traverse (do-traverse-hash (index value data) body)
           :pack     (values) ; we don't pack hashtables
           :capacity (values most-positive-fixnum)
           :count    (hash-table-count data))
```

Please see the online documentation for `defscheme`.

3.1 Representetion schemes

The current version of Spartns supports these representation schemes:

- `HASH`: may have good performance, depending on your Lisp implementation. Elements can be inserted in any order without affecting performance;
- `CVECTOR`: may be very fast, depending on your Lisp implementation. Inserting elements out of index order is very slow;
- `ARRAY`: theoretically, the fastest representation scheme (but hashtables can be faster, depending on your Lisp implementations). You can insert elements in any order, but the indices in each dimension *must* be from 0 to N-1, and you cannot change the dimension size.

For the two first schemes, the `:non-zero` field has the same meaning: the number of non-zero elements in that dimension. For `ARRAY`, it actually means the maximum capacity.

4 Performance

The choice of representation scheme depends a lot on your implementation of Lisp. The `hash` scheme can be faster or slower than `cvector`: on CLisp and ABCL, `hash` is faster; on SBCL and GCL, `cvector` is faster. I suggest that you run the micro-benchmarks that come with Spartns in order to determine what representation scheme will be the best (of course, your application is also a good benchmark!)

When you tell `defspartn` to create functions and macros for you, then the result of the `get-` function will be boxed. Using `w/spartns` defines local functions, so there doesn't need to be boxing (this probably depends on your Lisp implementation, actually... But it's true for SBCL).

Optimizing your use of Spartns can be summarized as follows.

- Start without the `declare` argument. The default is `(optimize (speed 3) (safety 1))`, which will allow you to check if the application is functioning correctly before you optimize. After that you can use `(safety 0)` if you want;
- Profiling Spartns is tricky. If you use `w/spartns`, your implementation may not profile the local calls. If you tell `defspartn` to generate `DEFUNs`, the overhead of the non-local function call will be profiled, when it wouldn't be present if you used `w/spartns`. Keep all this in mind, and do profile anyway;
- Check what representation scheme works best for you. Also check wether it's better to use GET/SET or to TRAVERSE the tensors;
- Once you are confident that your program is correct and that you have found the right representation scheme for your tensors, call `defspartn` with `def = NIL` and use `w/spartns`;
- If you know that you need a different representation scheme because of cache and locality issues, for example, you can add your own.

5 Tests

To run the tests, just load the +`do-tests.lisp` file. The tests are certainly not "for real" and need a lot of work, but they have been useful to detect some basic problems.

6 Internals

Spartns makes heavy use of macros, because it is basically a Lisp code generator. Because of this, understanding the code can be difficult for someone not used to writing "metacode".

6.1 Representations

Each mapping from fixnums onto objects can be represented in several different ways. These are called *representation schemes*, and each scheme is described in a structure and stored in a hashtable in the special variable `*representation-schemes*`.

Representation schemes are defined using the `defscheme` macro, which stores the description of the scheme in the scheme database.

This:

```
(defscheme :name      hash
           :type      hash-table
           :make      (make-hash-table :size size)
           :get       (gethash index data sparse-element)
           :set       (setf (gethash index data) value)
           :delete    (remhash index data)
           :traverse (do-traverse-hash (index value data) body)
           :capacity (values most-positive-fixnum)
           :count     (hash-table-count data))
```

Will create a hashtable that maps `NAME` to `HASH`, `TYPE` to `HASH-TABLE`, and so on, and add this to the hashtable in the variable `*representation-schemes*`, under the key `HASH`.

Representation schemes can map fixnums onto objects only, so they can only be used for one dimension. The functions that operate on representation schemes can be combined in order to access structures composed of several dimensions. These are the chained-functions (`chained-get`, `chained-set`, etc).

The chained-functions produce textual Lisp code that can be used to access tensors of an arbitrary number of dimensions. This code is used by the functions `define-spartn-*`, which will produce the full `DEFUN` form.

The `defspartn` macro adds a spartn scheme to the hashtable in `*spartn-schemes*` and optionally calls `DEFUN` and `DEFMACRO` to create the functions and macros to access the new scheme.

As an example,

```
(defspartn my-scheme
           :representation (spartns:hash spartns:cvector)
           :non-zero       (10 20)
           :element-type   single-float
           :sparse-element 0.0
           :def            T
           :declare        (optimize (speed 3) (safety 0)))
```

Is expanded as:

```
(DEFUN make-my-scheme () ...)
(DEFUN get-my-scheme (data index-1 index-2) ...)
```

```
(DEFUN set-my-scheme (data index-2 index-2 value) ...)
(DEFUN delete-my-scheme (data index-1 index-2) ...)
(DEFUN copy-my-scheme (data) ...)
(DEFMACRO traverse-myscheme (index-list value data &body body) ...)
```

And has the side-effect of adding this structure:

```
#S(SPARTNS:SPARTN-SCHEME
   :representation (spartns:hash spartns:cvector)
   :non-zero       (10 20)
   :element-type   single-float
   :sparse-element 0.0
   :declare        (optimize (speed 3) (safety 0)))
```

to the hashtable in the special variable `*spartn-schemes*`.

However, `defspartn` would have *not* expanded as the series of `DEFUN`s and `DEFMACRO` if the `:def` key parameter was set to `NIL` (but it would still have the side effect of adding the scheme to `*spartn-schemes*`.

The macro `w/spartns` retrieves a spartn scheme from the database and creates local functions and macros using `LABELS` and `MACROLET`.

```
(w/spartns (my-scheme)
  (let ((mat (make-my-scheme)))
    (set-my-scheme mat 1 2 5.5)))
```

Is expanded as:

```
(LABELS ((make-my-scheme () ...)
         (get-my-scheme (data index-1 index-2) ...)
         (set-my-scheme (data index-1 index-2 value) ...)
         (delete-my-scheme (data index-1 index-2) ...)
         (copy-my-scheme (data) ...)
  (MACROLET ((traverse-myscheme
               (index-list value data &body body) ...)
    (set-my-scheme mat 1 2 5.5)))
```

6.2 Coding style

The style used in Spartns is not conventional in some aspects.

I use a large monitor and like to see as many lines of code as possible, so I do not break lines at 72 columns. I will probably reformat Spartns so more people can comfortably read the code.

I use key arguments in functions where there are several parameters and there is non-trivial recursion, because it makes the recursive call easier to read. The `chained-*` functions are like this.

The `expand-scheme` function can look a bit strange at first, but the example in the online documentation should help understand it.

One thing that I do agree to be ugly is the definition of binary-search as a macro (for efficiency). This was such a critical part of the code that I decided to just do that (inlining doesn't necessarily work, because the binary-search code needs type declarations – which the macro inserts). There is actually a function that will produce an optimized binary-search function, but it's not ready yet.

Another not-so-elegant thing is the overuse of `LET` and `LET*`. Sometimes it looks as if most of the work in a function is done inside the bindings of a `LET*` form (see `create-next-dim` for example).

I should also come up with a small meta-library for creating functions. Right now Spartns has lots of *ad-hoc* code to create several functions and macros.