Par&Con Final Project

Jay Kruer

May 12, 2021

Contents

1 Abstract

We implement a design for fast, parallel quadtree construction algorithm for a 2-dimensional `n`-body gravitation simulation à la Barnes-Hut. Our implementation features a web-based real-time renderer for simulations as well as a

graphical tool for constructing simulation scenarios. A full sequential implementation of Barnes-Hut and prototype parallel implementation of quadtree construction are provided. We hint at opportunities for further parallelization. Our work draws on a variety of sources ranging from the original paper of Barnes & Hut describing the sequential version of the algorithm to folkloreish bit-bashing traditions from the computer graphics community. Our approach for parallel quadtree construction makes large (almost 5x) performance improvements over the traditional sequential method.

2 Structure

- bhut.lisp: The main project code. A little under 600 lines of Common Lisp code.
- bhut.html: The client-side portion of the web-app.
- haskell-abandoned: our original sequential Barnes-Hut prototype, abandoned because debugging Haskell proved difficulty.
- report.pdf: This report.
- bhut-letter.pdf: one of our sources, link to in the references at the end of this file.

3 How to run it

The only program you need to install is SBCL, a compiler and interpreter for Common Lisp. On macOS you can install it with

```
brew install sbcl
```

With SBCL installed, you can run the program with

```
sbcl --load bhut.lisp
```

This should open a browser window pointed at `http://127.0.0.1:8080`. After that you can poke around and start simulating gravity.

4 Design

4.1 Barnes-Hut, sequentially

The idea of the Barnes-Hutt algorithm is relatively straight forward. If we want to compute the force of gravity on one object in space due to a bunch of other objects in space, we can squint and tree all the other objects as one big lump, provided they are far enough away for our squinting to be justified. A very careful form of squinting at objects for force computations was developed by Barnes and Hut in their 1986 *Letter to Nature*. This interactive tutorial gives a phenomenal visual overview of the approach: `https://jheer.github.io/barnes-hut/`. Barnes and Hut built on work done by Andrew Appel as an undergraduate and put the approximation algorithm he developed on better quantitative footing for use in calculations more important than just watching bodies fly around. Their approach hinges on a clever data structure: the quadtree. Below we give our Common Lisp definition of the data structure. We chose to implement most of our data structures as heavy-weight CLOS classes (as opposed to more primitive structs also featured in the language) because they allow extensive introspection at runtime. This proved instrumental

in our development while ironing out bugs.

```
(defclass qtree ()
  ((body :initarg :body
         :initform nil) ;; the body contained in this node, nil unless the
                        ;; quadtree is a leaf

   (extent :initarg :extent
           :initform nil)    ;; a list '(xmin xmax ymin max) which keeps
                             ;; track of the area a given qtree node
                             ;; covers

   ;; the mass of all the objects contained within the extent of this tree
   (treemass
    :initarg :treemass
    :initform nil)
   ;; in the case of a (non-empty) leaf, this is just the object mass

   ;; center of mass of the extent represented by this tree
   (treecenter
    :initarg :treecenter
```

```
    :initform nil)

   ;; quadrants
   (q1 :initarg :q1 :initform nil)
   (q2 :initarg :q2 :initform nil)
   (q3 :initarg :q3 :initform nil)
   (q4 :initarg :q4 :initform nil)))
```

A quadtree node is comprised of at most one associated body, an extent specifying the region of $\mathbb{R}^2$ covered by the quadtree, and (optionally) four quadrant subtrees. A quadtree node covers some region of space, while a quadtree leaves covers at most a single point-mass. The approach taken by Barnes-Hut hinges on building a quadtree containing all the bodies you're interested in for your force computation. We give our sequential quadtree construction code below. We elide some of our more obvious subroutines in the interest of brevity. The core of the algorithm is the `insert-body` method of the quadtree class. It inserts a body into an existing quadtree with the appropriate extent by either 1) just setting the `body` field if the quadtree is an empty leaf; 2) quartering the quadtree if the tree is a leaf already containing a body; or 3) recursing on the subtree corresponding to the appropriate quadrant in the case of a non-leaf node tree.

```
(defmethod quarter ((tree qtree))
  "Splits a full leaf (one with a body) into quadrants, placing its body into
   the appropriate quadrant." (elided))

(defmacro insertff (body trees)
  "Insert a body into the first-fitting subtree. This is a macro to help SBCL
   optimize the mutual tail-recursion."
  `(loop for tree in ,trees do
    (with-slots (extent) tree
      (if (inextp ,body extent)
          (return (insert-body tree ,body))))))

(defmethod insert-body ((tree qtree) b)
  (with-slots (body extent treemass treecenter q1 q2 q3 q4) tree
    (if (inextp b extent)
        (if (not body)
            (if (leafp tree)            ; just fill the empty leaf
                (progn
```

```
                  (setf body b)
                  (setf treemass (slot-value b 'mass))
                  (setf treecenter (slot-value b 'position)))
                                ; here we have a node already split into
                                ; quadtrees, so we just need to insert b
                                ; in to the first fitting subtree and
                                ; update all the tree metrics
                                ; appropriately
                (progn (setf treecenter (compcenter (cons b (allbodies tree))))
                       (setf treemass (compmass (cons b (allbodies tree))))
                       (insertff b `(,q1 ,q2 ,q3 ,q4))))


                                ; full leaf: in this case we need to
                                ; split the leaf, as we only allow one
                                ; body per leaf
            (if (not (equalp body b))
                (progn
                  (quarter tree)
                  (setf treecenter (compcenter (cons b (allbodies tree))))
                  (setf treemass (compmass (cons b (allbodies tree))))
                  (insertff b `(,q1 ,q2 ,q3 ,q4)))))
        (error "Tried inserting a body into a tree with wrong extent"))))

(defun build-qtree (bodies)
  (let ((init-tree (make-qtree (compextent bodies))))
    (progn
      (loop for bod in bodies do
        (insert-body init-tree bod))
      init-tree)))
```

With this, the force computation is a straightforward translation of the original code given in (Barnes & Hut 1987). We diverge slightly from their presentation by treating force as primary in our calculation and deriving acceleration and position changes elsewhere in the program. The algorithm proceeds as follows: to compute the total force on one of the bodies in our simulation due to every other body we traverse the whole quadtree containing the bodies, summing up the force due each subtree until, in the base case, for trees far enough away from the point at hand, we approximate the force due to the tree's bodies by computing the Newtonian force on the body due

to the *centroid* of that tree. The centroid is simply the sum mass of the bodies contained in the tree positioned at the center of mass of the tree's bodies. The sensitivity θ is used in determining whether a tree is small enough and close enough to be collapsed to its centroid for the purpose of force calculation and is usually set to 1. This setting has suited our purposes well in the prototype.

```
(defmethod centroid ((tree qtree))
  (with-slots (treecenter treemass) tree
    (make-instance 'body :mass treemass :position treecenter :velocity #(0 0))))

(defun comp-force (body tree)
  (with-slots (extent treecenter treemass q1 q2 q3 q4) tree
    (if (or (emptyp tree) (not extent))
        #(0 0)
        (if (not treecenter)
            (error "Trying to compute force due to tree with no center")
            (let* ((xmin (car extent))
                   (xmax (cadr extent))
                   (ymin (caddr extent))
                   (ymax (cadddr extent))
                   (diameter (euclidean-dist treecenter (slot-value body 'position)))
                   (cell-length (max (abs (- xmin xmax)) (abs (- ymin ymax)))))
              (if (or (equalp diameter 0.0)
                      (< (/ cell-length diameter) *theta*))
                  (newtonian-force body (tree-body tree))
                  (reduce #'v+ (mapcar #'(lambda (subtree) (comp-force body subtree))
                                       (subtrees tree)))))))))
```

4.2 Parallelizing Barnes-Hut

Barnes-Hut's technique for gravity simulation has several stages, and it's not clear a priori where once should focus their parallelization efforts. The obvious easy approach is to simply parallelize the force computations: one could just assign disjoint groups of the bodies to workers/threads and then perform the force calculations sequentially for each group in every worker. Parallelizing the other major component of the Barnes-Hut procedure, namely quadtree construction, is trickier.

If given a list of bodies ordered in no particular way, there is no obvious way to divide up the work of quadtree construction and still obtain a sensible

quadtree for force computations. One clever way to divide up the work is through the use of a **space-filling curve** called a **Morton curve** or **Z-order curve**. Here's the idea: we cleverly assign to each point in $\mathbb{R}^2$ a value in $\mathbb{N}$, called the point's **Morton code**. We do this in such away that when sorting with respect to the usual order on $\mathbb{N}$ a sequence of points keyed by their Morton codes, we have that any 4-partition of the sorted sequence corresponds to a disjoint quartering of the plane. The upshot is that we may independently construct quadtrees for each of the four subsequences and then trivially reassemble them into a quadtree covering all of the bodies. The wakeful reader will point out that using a comparison based-sort ($\Omega(n \log n)$) here wouldn't result in any asymptotic improvements for our algorithm, since we would already do at least as much work as sequential Barnes-Hut in just the sorting phase. It turns out that we can use a binary radix sort to sort by Morton-codes, as we do in our prototype implementation, which results in favorably asymptotic behavior for our algorithm provided points are close enough to the origin. Some tweaking left to future work may eliminate this caveat. A visualization of the Morton curve is given below. A common concrete Morton order is attained by interleaving the bits of each coordinate of a point. We point to an efficient method for the bit-interleaving below in the implementation section.

For lack of a proof of the correctness this method, we have conducted empirical tests of the method using our prototype parallel implementation. Our test involved shuffling the input bodies randomly before applying our Morton-code-based parallel implementation quadtree construction and then proceeding to the force computations. The observed results seem about right; a more scientific test would directly compare the forces computing when using the quadtrees produced by the parallel and sequential implementations when given the same list of shuffled bodies.

4.3 Proposed improvements

4.3.1 Parallel force computation

We currently perform the force computation for each body in sequence. This operation is trivially parallelizable with a parallel-map construct and would likely result in substantial time performance gains. We leave that performance on the table, for now.

5 Implementation

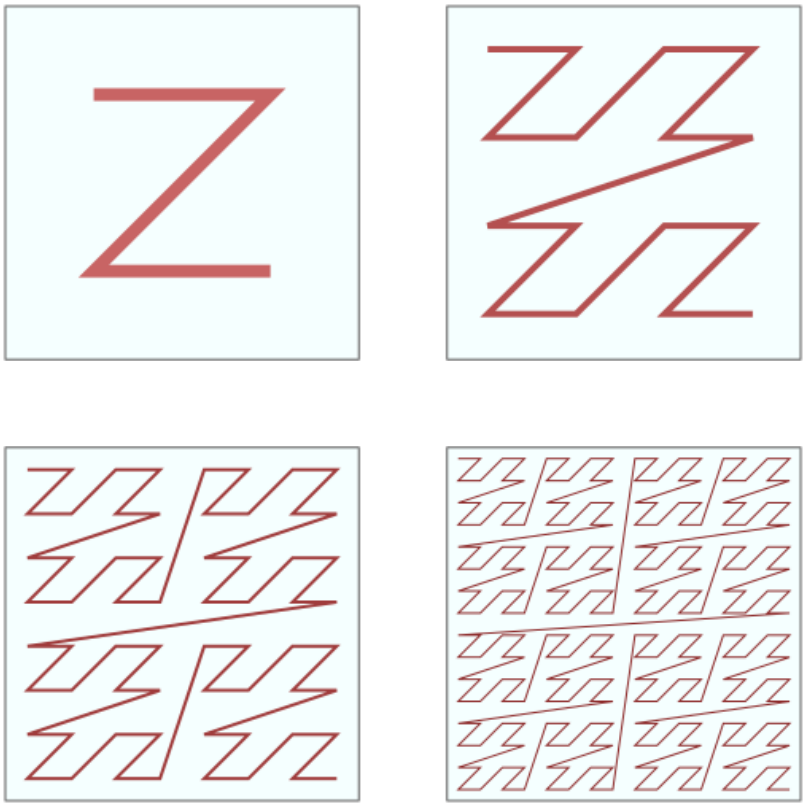

Figure 1: Illustration of iterated computation of a Z-ordering on $\mathbb{R}^2$: By David Eppstein, based on a image by Hesperian. CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=3879675`

5.1 Computing Morton codes

Our Morton code computation is a simple transcription of the following C routine to a Lispy format: `http://www.graphics.stanford.edu/~seander/bithacks.html#InterleaveBMN`. Perhaps surprisingly, Common Lisp has very strong support for bit-bashing. Our code is written below. A brief discussion of its operation follows.

```
(defun intersperse-hack (x y)
  "intersperses the bits of x and y by some fancy bit-bashing

   borrowed from http://www.graphics.stanford.edu/~seander/bithacks.html#InterleaveBMN
   return by date: Common Lisp EOL"
  (let ((B '(#x55555555 #x33333333 #x0F0F0F0F #x00FF00FF))
        (S '(1 2 4 8))
        (x x)
        (y y))
    (setf x (logand (logior x (ash x (nth 3 S)))
                    (nth 3 B)))
    (setf x (logand (logior x (ash x (nth 2 S)))
                    (nth 2 B)))
    (setf x (logand (logior x (ash x (nth 1 S)))
                    (nth 1 B)))
    (setf x (logand (logior x (ash x (nth 0 S)))
                    (nth 0 B)))

    (setf y (logand (logior y (ash y (nth 3 S)))
                    (nth 3 B)))
    (setf y (logand (logior y (ash y (nth 2 S)))
                    (nth 2 B)))
    (setf y (logand (logior y (ash y (nth 1 S)))
                    (nth 1 B)))
    (setf y (logand (logior y (ash y (nth 0 S)))
                    (nth 0 B)))
    (logior x (ash y 1))))
```

The routine interleaves the bits of x and y (unsigned integers in the range [0,65536=2^16] playing the part of of bit-arrays) in a new 32-bit integer. The size restrictions on x and y are there since we only have 32 bits of room in z; we can only interleave the bits of numbers with at most 16 significant bits.In

the interest of making things slightly less opaque, the magic numbers stored in `B` are rendered in binary as:

- `0x55555555` = `0b1010101 01010101 01010101 01010101`
- `0x33333333` = `0b0110011 00110011 00110011 00110011`
- `0x0F0F0F0F` = `0b0001111 00001111 00001111 00001111`
- `0x00FF00FF` = `0b0000000 11111111 00000000 11111111`

the first maybe hinting at how this works: logically `$AND$`ing with that bit-vector would pick off the odd bits of a number. In the project, we took this code as a black-box after performing some preliminary tests to verify its behavior. We admit that we still don't fully understand the inner workings of this routine!

5.2 Coarse parallelization with pmap

We implemented the parallelization scheme described above using a high-level parallelism library for Common Lisp called `lparallel`. The library's primary offerings are parallel versions of well-loved Lisp list combinators like `map` and `reduce`. We found these primitives very easy to use for the actual parallel computation involved in parallel quadtree construction. As mentioned in the design section, the actual parallelism involved in our strategy is relatively straight-forward; the meat of the work in parallelizing quadtree construction occurs in Morton-sorting the bodies and then partitioning the sorted bodies into a set of chunks associated with each worker. This work is done in the `schedule` function, which is elided because it won't fit horizontally on the page. This function is among the most complicated in the implementation, but all one really needs to know is that it takes a list of bodies, and outputs a "top-level" quadtree as well as chunks of the bodies together with *associated* quadtree objects. We orchestrate the construction of the top-level tree and chunk-trees in such away that filling in all the quadtrees for the chunks results in a full top-level quadtree. The power of mutability! If the bodies are additionally Morton-ordered, then the quadtree is also well-formed and usable for force computations. The parallel counterpart to `quadtree` is shown below, but to really understand how its working one should check out the `schedule` code.

```
(defun par-build-qtree (bodies)
  (let ((sorted (morton-sort bodies)))
```

```
    (destructuring-bind (top . chunks)
                        (schedule sorted
                                  (cl-cpus:get-number-of-processors))
      (let ((body-array (map 'vector #'identity sorted)))
        (lparallel:pmap 'nil
                        #'(lambda (chunk)
                            (destructuring-bind (bs tree range) chunk
                              (declare (ignore bs))
                              (finish-build tree range body-array))) chunks))
      top)))
```

5.3 Evaluation

We benchmarked our implementation on a test set of a little over fifteen-thousand bodies, whose definition is shown below. As seen from the output below, on this example we observe about a 5x speed improvement over the sequential implementation for our parallel quadtree construction procedure when run on a system with 12 processors (Intel Core i7 9750H). We expect there is a lot of performance left on the table by various implementation choices we made in the interest of timely completion of the project. See the proposed improvements section below for more details.

```
(defparameter *test-bodies*
  (loop for x from 0 to 150
        append (loop for y from 0 to 100
                     collect (make-instance 'body :position `#(,(coerce x 'float) ,(co
                                                  :mass 1
                                                  :velocity #(0 0)))))

CL-USER> (time (build-qtree *test-bodies*))
Evaluation took:
  99.250 seconds of real time
  103.723704 seconds of total run time (97.329841 user, 6.393863 system)
  [ Run times consist of 14.816 seconds GC time, and 88.908 seconds non-GC time. ]
  104.51% CPU
  257,255,099,802 processor cycles
  104,270,459,520 bytes consed
```

```
CL-USER> (time (par-build-qtree *test-bodies*))
Evaluation took:
  18.442 seconds of real time
  33.669276 seconds of total run time (31.087524 user, 2.581752 system)
  [ Run times consist of 3.553 seconds GC time, and 30.117 seconds non-GC time. ]
  182.57% CPU
  47,802,837,076 processor cycles
  16,514,741,584 bytes consed
```

Real world performance can also be evaluated with the web-app, launched by evaluating `sbcl --load bhut.lisp` from the project root directory. It should handle everything. There is a lingering floating point comparison bug in `insert-body` that seems to preclude placing bodies right on top of each other during initial placement; be careful to space the bodies out. We have noted that the performance of our parallel implementation is worse than that of the sequential implementation for live-rendered low-body scenarios. Considering the benchmark above, our implementation would be more suited to pre-computed simulations of scenarios enlarging massive numbers of bodies. We would eventually like to add very large simulations to our web-app, potentially paired with a buffering option to allow for faster rendering.

5.4 Proposed improvements

5.4.1 Parallel radix sort

We currently use a sequential radix sort for Morton sorting. This was sufficient for our implementation of parallel quadtree construction as described above to offer considerable speed improvements over the sequential version for large number of bodies. We conjecture that speeding up the radix sort through parallelism might gain us enough performance improvement to beat the sequential version in even small simulation scenarios.

5.4.2 Direct pthread implementation

In the interest of saving time, our prototype uses a high-level parallelism library for Common Lisp called `lparallel`. Rather than going all the way to CUDA, a more conservative option for increasing the performance of our implementation would be to replace the calls to `lparallel:pmap` with hand-coded parallel routines using `pthreads`. Common Lisp does not have a standardized interface to `pthreads`, but the `bordeaux-threads` library offers a lightweight and, crucially, portable abstraction over `pthreads`.

5.4.3 CUDA rewrite

We originally intended to implement the parallel quadtree construction with NVIDIA's CUDA, or a similar technology like WebGPU or OpenCL. This would constitute a fairly invasive modification to our code; defining the necessary kernel would amount to rewriting our body insertion algorithm in a C-like DSL which we simply didn't have time for. That said, the large number of cores made available by GPUs would present an opportunity for massive time savings with the same design. We plan to pursue this option this summer :)

6 Collected references

Each of the bullets below features a clickable link.

- The original Barnes-Hut paper.
- NVIDIA developer blog on parallel tree construction: This blog inspired us to go check out Morton sorting for the parallelization. They use considerably more advanced techniques in their parallel quadtree construction, including a GPU-based parallel radix sort.
- Another blog on Morton codes which pointed us to the bitbashing method for doing bit-interleavings
- The bit-bashing used for fast bit interleaving
- An interactive tutorial on quadtrees and Barnes-Hut which we found useful during initial investigations.
- Another great quadtree visualization article.