Rewrite the Problem-Based Benchmark Suite in Elixir

Luiz Berto
UFMG
Brazil
luiz.berto@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG
Brazil
fernando@dcc.ufmg.br

Abstract

In this work, we tackle the rewrite of the Problem Based Benchmark Suite (PBBS) in the Elixir programming language. Elixir is a dynamically-typed, functional language, whose concurrency model is based on the actor model. We wrote parallel and sequential implementations for 8 of the 22 problems of the PBBS, and experimentally evaluated the speedups achieved by the parallel implementations. We found that communication between Elixir's processes is the main bottleneck, preventing the parallel implementations from achieving larger speedups. For problems whose sequential algorithm time complexity is linear, speedups were negligible, because communication costs dominate the overall wall-clock time. As a corollary, for more expensive problems, where communication time is not the most time-consuming operation, higher speedups could be achieved.

Keywords: Parallel Programming, Actor Model

1 Introduction

This work aims to implement the *Problem Based Benchmark Suite (PBBS)* [3] problems in the Elixir programming language, in parallel. Elixir is functional, dynamically typed language, created in 2011 built on top of the Erlang Virtual Machine (BEAM). One of Elixir's main features is its adoption of the [1] actor model as a concurrency model, a feature inherited from Erlang in which all communication between different execution threads take place via message passing. This model is quite expressive, and differs substantially from the shared memory model, which is widely adopted by multiple mainstream programming languages such as C and Java. One of the main objectives of this work is to exercise the parallel programming capacity of the language using the actor model, and experimentally compare the performance of the parallel and sequential implementations written in Elixir (comparing the performance of different programming languages is a non-goal).

PBBS is a suite of problems created as a basis for benchmarking parallel implementations, and has 22 problems, subdivided into 5 categories:

- Basic building blocks:
  - comparison-based sort
  - histogram
  - remove duplicates
  - integer sort
- Graph algorithms:
  - breadth-first search
  - maximal independent set
  - maximal matching
  - minimum spanning forest
  - spanning forest
- Text processing:
  - Burrows-Wheeler decode
  - inverted index
  - longest repeated substring
  - suffix array
  - word counts
- Computational geometry/graphics:
  - convex hull
  - nearest neighbors
  - ray-triangle intersection
  - Delaunay triangulation
  - Delaunay refinement
  - range query 2D
- Other:
  - classification
  - n-body force calculation

Parallel and sequential algorithms have been implemented for 8 out of the 22 problems, using idiomatic Elixir code. For each problem, the parallel and sequential implementations were compared in a benchmark suite, varying inputs and parallelism levels where appropriate.

The rest of this paper is organized as follows: section 2 presents the theoretical references upon which this work is built on, section 3 explains each of the tackled problems and their implementations, section 4 presents the experiments and their results, and section 5 presents our conclusions.

2 Theoretical references

Many programming languages adopt the shared memory model as a concurrency model. This model allows concurrent programs to be written through direct use of threads and shared memory blocks. Generally, programs written under this model need to make use of synchronization primitives to ensure mutual exclusion of shared memory regions. In practice, it is desirable to perform as little mutual exclusion as possible so as not to generate large overheads, which in itself is a complicated task, in addition to the intrinsic difficulty of producing correct programs under this paradigm [4].

The actor model, on the other hand, works with the concept of message passing. In this model, the primary computational entity is called an actor, and actors do not share any

memory regions with each other. Instead, all communication takes place through exclusive communication channels, where an actor can send a direct message to another actor, which receives it in a "mailbox", in an asynchronous and reactive manner. The underlying *runtime* guarantees message delivery, making the actor model in practice an abstraction for writing concurrent and parallel programs. The actor model is very similar to the object-oriented paradigm [7], which is also based on the concept of *message-passing*, which means that Elixir, being a functional programming language, can benefit from the advantages both of these paradigms.

The work by Blelloch [3] proposes a framework for comparing parallel implementations, and this work aims to be an implementation of the problems in Elixir.

3 Problems and implementations

In this section, the problems implemented in this work will be discussed in further detail. Each problem will be presented, as well as a sequential algorithm and at least one parallel algorithm, which were implemented in Elixir. For convenience, instead of working directly with the *send* and *receive* primitives, all parallel implementations use either *Task.async* or *Task.async_stream* to perform parallel computations. These functions are convenience wrappers that spawn an actor (or process in Elixir jargon) that receives an input message, performs some kind of operation, and send a message back to the calling actor with the output.

3.1 Histogram

3.1.1 The problem. Let A be a list of n of positive integers in the range $[0, m]$ (m is a parameter of the algorithm). The goal is to compute the frequencies of each integer from 0 to m in A, and output a list with $m + 1$ elements, where each position i represents the frequency of the value i in the input list A.

For example, if $A = [1, 1, 1, 3, 3, 4]$ and $m = 4$, then the output should be $[0, 3, 0, 2, 1]$:

- the first position, which represents the frequency of the number 0 in the input array, is 0, as 0 does not appear in the input array
- the second position, which represents the frequency of the number 1 in the input array, is 3
- and so forth

3.1.2 Sequential algorithm. The sequential algorithm uses *Enum.frequencies*, a function that counts the frequencies of elements in a list and returns a key-value data structure. There is no guarantee for the order of keys, so an extra step is needed to map the values to a list in correct order. This is done by traversing the range $[0, m]$ in order, and creating a list of the frequency values.

The implementation is as follows:

```
def histogram(nums, buckets) do
  map = Enum.frequencies(nums)

  Enum.map(0..buckets, fn bucket ->
    Map.get(map, bucket, 0)
  end)
end
```

3.1.3 Parallel algorithm. The parallel algorithm divides the input list in parts, and delegates the computation of the frequencies of each part to one of p (a parameter that controls the parallelism degree) workers that run in parallel. This strategy, the *chunking* strategy, is also used in other problems.

The division of the input data is achieved through the chunk_every function from the *Stream* module. This function chunks the input *list* into chunks of size *chunk_size*, and does that in a lazy fashion (returning a *Stream*), which is important to prevent a large sequential section in the main process. 20000 was chosen as the chunk size after experimentation, but different workloads might benefit from different values. It is important to note that more than p chunks can be created depending on the input size, but *Task.async_stream* limits the maximum number of parallel workers.

The delegation to workers happens via *Task.async_stream*, which is a function that accepts an enumerable (*Stream* and *List* are both enumerables), applies a given function on each item of the enumerable in parallel, with at most p workers, and returns the resulting enumerable. Each worker returns a map, and the main process accumulates each map into a single map, merging them by summing equal keyed values. By the end, the result list is created using the same strategy as the sequential algorithm.

```
def histogram(nums, buckets, p) do
  chunk_size = 20000

  map =
    Stream.chunk_every(nums, chunk_size)
    |> Task.async_stream(
      fn elements ->
        Enum.frequencies(elements)
      end,
      max_parallelism: p - 1,
      ordered: false,
      timeout: :infinity
    )
    |> Stream.map(fn {:ok, v} -> v end)
    |> Enum.reduce(%{}, fn res, acc ->
      Map.merge(res, acc, fn _key, v1, v2 -> v1 +
    v2 end)
    end)

  Enum.map(0..buckets, fn bucket ->
    Map.get(map, bucket, 0)
  end)
end
```

3.2 Remove Duplicates

3.2.1 The problem. Let A be a list of integers. The goal is to remove all duplicates from A, and return a list with no duplicates. There is no constraint regarding the output order.

3.2.2 Sequential algorithm. The sequential algorithm simply builds a set data structure out of the input list, and converts the set back to a list:

```
def remove_duplicates(nums) do
  MapSet.new(nums)
  |> Enum.to_list
end
```

3.2.3 Parallel algorithm. The parallel algorithm uses the same general strategy as the parallel histogram algorithm: work is evenly delegated to p workers, and at the end the results are combined. However, instead of using *Streams* and *Task.async_stream* to delegate to workers, it broadcasts the input list by using *ets*. *ets* is short for Erlang Term Storage, which is a shared memory region within the Erlang VM that all processes can access. In this context, the use of *ets* can be seen as an alternative way to broadcast a message to multiple processes at the same time: as all processes can see the data in *ets*, there is no need for the main process to sequentially send the same message multiple times, for each worker. The main process writes the entire input list to *ets*, and workers fetch the input list to their own stacks. As each worker has to process only part of the input list, the combination of *Enum.drop* and *Enum.take_every* guarantees equal workload. This strategy was preferred as opposed to *Task.async_stream* after showing better results in experiments.

Each worker simply creates a new set from their part of the list, and at the end, the main process combines the sets with union operations, and creates the output list. The implementation is as follows:

```
def remove_duplicates(nums, p) do
  :ets.new(:ddup, [:public, :named_table])
  :ets.insert(:ddup, {:data, nums})

  ret =
    0..(p - 1)
    |> Enum.map(fn i ->
      Task.async(fn ->
        input = Keyword.get(:ets.lookup(:ddup, :data), :data)

        Enum.drop(input, i)
        |> Enum.take_every(p)
        |> MapSet.new()
      end)
    end)
    |> Task.await_many()
    |> Enum.reduce(MapSet.new(), fn res, acc ->
      MapSet.union(res, acc)
    end)

  :ets.delete(:ddup)

  Enum.to_list(ret)
end
```

3.3 Integer Sort

3.3.1 The problem. Let A be a list of n of positive integers in the range $[0, n)$. The goal is to sort this list using an integer sorting algorithm, such as counting sort or radix sort.

3.3.2 Sequential implementation. The chosen algorithm for this problem is the radix sort algorithm. The sequential implementation buckets the numbers from the input list from the least significant digit to the most significant digit (in base 10). This goes on until the base case is hit: when the recursive call tries to bucket by a digit that does not differentiate any items from the list, meaning that the list is already sorted. The implementation follows:

```
def radix_sort([], _), do: []

def radix_sort(list) do
  max = abs(Enum.max_by(list, &abs(&1)))
  sorted_list = radix_sort(list, max, 1)

  {negative, positive} = Enum.split_with(
    sorted_list, &(&1 < 0))
  Enum.reverse(negative, positive)
end

defp radix_sort(list, max, m) when max < m, do:
    list

defp radix_sort(list, max, m) do
  buckets = List.to_tuple(for _ <- 0..9, do: [])

  buckets =
    Enum.reduce(list, buckets, fn item, acc ->
      index = abs(item) |> div(m) |> rem(10)
      put_elem(acc, index, [item | elem(acc, index
    )])
    end)

  sorted_by_digit = Enum.reduce(9..0, [], fn i,
    acc -> Enum.reverse(elem(buckets, i), acc) end
    )

  radix_sort(sorted_by_digit, max, m * 10)
end
```

3.3.3 Parallel implementation. The parallel implementation uses the *chunking* strategy. Each worker calls the sequential algorithm shown above to order its sublist, and at the end all parts are combined using a merge operation:

```
def radix_sort(list, p) do
  chunk_size = 20000

```

```
  Stream.chunk_every(list, chunk_size)
  |> Task.async_stream(
    fn elements ->
      PBBS.Sequences.IntegerSort.Sequential.
    radix_sort(elements)
    end,
    max_parallelism: p - 1,
    ordered: false,
    timeout: :infinity
  )
  |> Stream.map(fn {:ok, v} -> v end)
  |> Enum.reduce([], fn item, acc ->
    [item | acc]
  end)
  |> :lists.merge()
end
```

3.4 Comparison Sort

3.4.1 The problem. Let A be a list of n arbitrary elements. The goal is to sort this list using any comparison-based sorting algorithm.

3.4.2 Sequential algorithm. The chosen algorithm is the merge sort, which is built-in to the language in *Enum.sort*.

3.4.3 Parallel algorithm. The parallel implementation also uses the *chunking* strategy, with each worker using *Enum.sort* to order its sublist:

```
def merge_sort(list, p) do
  chunk_size = 20000

  Stream.chunk_every(list, chunk_size)
  |> Task.async_stream(
    fn elements ->
      Enum.sort(elements)
    end,
    max_parallelism: p - 1,
    ordered: false,
    timeout: :infinity
  )
  |> Stream.map(fn {:ok, v} -> v end)
  |> Enum.reduce([], fn item, acc ->
    [item | acc]
  end)
  |> :lists.merge()
end
```

3.5 Word Count

3.5.1 The problem. Let S be a string. The goal is to count the occurrences of the words in S, normalizing them to lower case, and replacing all non-letter characters with whitespace. For example, if S ="Alpha 1 ALPHA beTa beta3beta", then the algorithm should consider that both "alpha" and "beta" appear twice, and no other word occurs in S.

3.5.2 Sequential implementation. The sequential implementation uses a regular expression to split the input string into words, whenever non-letter characters are found. Then, strings are lowered, empty strings are filtered, and the words occurrences are counted with *Enum.frequencies*:

```
def word_count(string) do
  String.split(string, ~r/[^A-Za-z]+/)
  |> Enum.map(&String.downcase/1)
  |> Enum.filter(fn s -> s != "" end)
  |> Enum.frequencies()
end
```

3.5.3 Parallel implementation. The parallel implementation also uses the *chunking* strategy, leveraging *String.splitter* to lazily split the input string on whitespaces. Then, each worker uses the same regular expression used in the sequential implementation to filter out any non-letter characters, and proceed to lower, filter empty strings, and count their frequencies. At the end, the frequency maps are merged to compose the final output:

```
def word_count(string, p) do
  workload = 1000

  String.splitter(string, :binary.compile_pattern
    ([" ", "\n", "\t", "\f", "\r"]))
  |> Stream.chunk_every(workload)
  |> Task.async_stream(fn part ->
    word_count_internal(part) end,
    max_concurrency: p,
    ordered: false,
    timeout: :infinity
  )
  |> Stream.map(fn {:ok, v} -> v end)
  |> Enum.reduce(%{}, fn elem, acc ->
    Map.merge(elem, acc, fn _, a, b -> a + b end)
  end)
end

def word_count_internal(input) do
  Enum.flat_map(input, fn s -> String.split(s, ~r
    /[^A-Za-z]+/) end)
  |> Enum.filter(fn s -> s != "" end)
  |> Enum.map(&String.downcase/1)
  |> Enum.frequencies()
end
```

3.6 Suffix Array

3.6.1 The problem. Let $S = S_1S_2...S_n$ be a string. A substring of S is a contiguous set of characters occurring in S, identified by start and end positions, such as $S_{1-3} = S_1S_2S_3$. A suffix of S is a substring of S whose ending position is n, such as $S_2 = S_2S_3...S_n$. By definition, S has n suffixes, each of which is identified by its starting position (by S_2, we are referring to the suffix that starts at the second character of S).

The *suffix array* of S is an array that stores the identifiers of the suffixes of S, sorted in ascending lexicographic order.

For example, if $S = bdca$, we have the following suffixes:

- $S_1 = bdca$
- $S_2 = dca$
- $S_3 = ca$
- $S_4 = a$

If we create a list with the suffixes of S, and sort it in ascending lexicographic order, we have the following result: $[a, bdca, ca, dca]$. The suffix array is nothing more than this ordered list, but instead of keeping the suffixes themselves, only their identifiers are stored. Therefore, the suffix array of $S = bdca$ is: $[4, 1, 3, 2]$.

Suffix array is a data structure widely used in computational biology contexts [8], and in such contexts it is common to deal with very long DNA strings, therefore the existence of efficient suffix arrays construction algorithms is desirable.

Here, we are interested in the construction algorithm of a suffix array also known as *SACA* (suffix array construction algorithm) in the literature [6]. The input of a *SACA* is a string S, and the output is the suffix array of S.

3.6.2 Sequential implementation.

As a basis for comparison, a naive, sequential suffix array construction algorithm was implemented. This algorithm simply generates all suffixes of S (with their respective indices), sorts them in ascending lexicographic order, and returns only the indices of the resulting array.

The implementation of this algorithm in Elixir is as follows:

```
def suffix_array(string) do
    len = String.length(string)
    suffixes_start_indexes = 0..(len - 1)

    suffixes = Enum.map(
      suffixes_start_indexes,
      fn s -> {s, binary_part(string, s, len - s)}
     end
    )

    Enum.sort_by(suffixes, &elem(&1, 1))
    |> Enum.map(&elem(&1, 0))
end
```

One can see the great expressiveness of the language, which allows us to implement the algorithm in only 12 lines.

3.6.3 Parallel implementation.

The parallel version of the naive algorithm splits the S suffixes into p parts, which are sorted in parallel, and merged sequentially in a final operation. p is the parameter which controls the degree of parallelism of the algorithm, and in the experiments, different values for p were tested. The implementation of this modification of the naive algorithm is as follows:

```
def suffix_array(string, p) do
    len = String.length(string)
    size = div(len, p)
    parts = 0..(len-1) // size

    tasks = Enum.map(parts, fn start ->
      Task.async(fn ->
        start_indexes = start..(min(start + size - 1, len - 1))
        suffixes = Enum.map(
          start_indexes,
          fn s -> {s, binary_part(string, s, len - s)} end
        )
        sorted = Enum.sort_by(suffixes, &elem(&1, 1))

        Enum.map(sorted, fn ({index, suffix}) -> {suffix, index} end)
      end)
    end)
    results = Task.await_many(tasks, :infinity)
    merged = :lists.merge(results)

    merged
    |> Enum.map(&elem(&1, 1))
end
```

3.7 Convex Hull

3.7.1 The problem.

Let P be a set of 2-dimensional points. The convex hull of P is the smallest convex polygon that contains all points in P. In this case, we are interested in the vertices of such polygon, which by definition must all be contained in P. The output should be the indexes of the input points that compose the convex hull, in clockwise order, starting from its leftmost point (minimum x coordinate).

3.7.2 Sequential implementation.

The algorithm chosen for the sequential implementation is the *QuickHull* algorithm [2], which is a recursive algorithm. Initially, *QuickHull* finds the leftmost (L) and rightmost (R) points on the set, and divide the points by their orientation with respect to the segment LR: points that lie on the left of LR, and points that lie on the right of LR.

Then, each part is processed separately. For each part, the algorithm finds the point X that is furthest from the LR segment (by definition, this point is part of the convex hull). After that, the algorithm calls itself recursively for the segments LX and XR, and concatenates both results. The base case for the recursion is when only one point is at one side of a segment: this point must also be part of the convex hull. The implementation is as follows:

```
def convex_hull(points) do
  indexed_points = Enum.with_index(points)
  min_x = Enum.min_by(indexed_points, fn ({point, _idx}) -> elem(point, 0) end)
  max_x = Enum.max_by(indexed_points, fn ({point, _idx}) -> elem(point, 0) end)

```

```
  hsplit(indexed_points, min_x, max_x) ++ hsplit(
    indexed_points, max_x, min_x)
end

def hsplit(points, {p1, p1_index}, {p2, p2_index})
    do
  cross = Enum.map(points, fn ({pt, _idx}) ->
    cross_product(pt, p1, p2) end)

  zipped = Enum.zip(points, cross)
  packed = zipped
  |> Enum.filter(fn ({_point, cross}) -> cross > 0
     end)
  |> Enum.map(fn ({point, _cross}) -> point end)

  if length(packed) < 2 do
    [p1_index] ++ Enum.map(packed, fn ({_pt, idx})
     -> idx end)
  else
    pm = Enum.max_by(zipped, fn ({_point, cross})
    -> cross end)
    |> elem(0)

    hsplit(packed, {p1, p1_index}, pm) ++ hsplit(
    packed, pm, {p2, p2_index})
  end
end

def cross_product({ox, oy}, {vx, vy}, {wx, wy}) do
  (vx - ox) * (wy - oy) - (vy - oy) * (wx - ox)
end
```

3.7.3 Parallel implementation.

The parallel implementation computes each of the two recursive calls the *QuickHull* algorithm makes in parallel. There is a cutoff at 1000 points, where the remaining recursive calls happen sequentially, to prevent spawning new processes for very small sets of points. The leftmost and rightmost points in the first iteration are also computed in parallel:

```
def convex_hull(points) do
  indexed_points = Enum.with_index(points)

  :ets.new(:ch, [:public, :named_table])
  :ets.insert(:ch, {:points, indexed_points})

  [min_x, max_x] = Task.await_many([
    Task.async(fn ->
      pts = Keyword.get(:ets.lookup(:ch, :points),
     :points)
      Enum.min_by(pts, fn ({{x, _y}, _idx}) -> x
    end)
    end),
    Task.async(fn ->
      pts = Keyword.get(:ets.lookup(:ch, :points),
     :points)
      Enum.max_by(pts, fn ({{x, _y}, _idx}) -> x
    end)
    end),
  ])

  [first, second] = Task.await_many([
    Task.async(fn ->
      pts = Keyword.get(:ets.lookup(:ch, :points),
     :points)
      hsplit(2,pts, min_x, max_x)
    end),
    Task.async(fn ->
      pts = Keyword.get(:ets.lookup(:ch, :points),
     :points)
      hsplit(3,pts, max_x, min_x)
    end)
  ])

  result = first ++ second
  :ets.delete(:ch)
  result
end

def hsplit(index, points, {p1, p1_index}, {p2,
    p2_index}) do
  zipped = Enum.map(points, fn ({pt, idx}) -> {{pt
    , idx}, cross_product(pt, p1, p2)} end)

  packed = zipped
  |> Enum.filter(fn ({_point, cross}) -> cross > 0
     end)
  |> Enum.map(fn ({point, _cross}) -> point end)

  if length(packed) < 2 do
    [p1_index] ++ Enum.map(packed, fn ({_pt, index
    }) -> index end)
  else
    pm = Enum.max_by(zipped, fn ({_point, cross})
    -> cross end)
    |> elem(0)

    if length(packed) < 1000 do
      hsplit(index*2, packed, {p1, p1_index}, pm)
    ++ hsplit(index*2 + 1, packed, pm, {p2,
    p2_index})
    else
      :ets.insert(:ch, {String.to_atom("pts#{index
    }"), packed})
      [first, second] = Task.await_many([
        Task.async(fn ->
          pts = Keyword.get(:ets.lookup(:ch,
    String.to_atom("pts#{index}")), String.to_atom
    ("pts#{index}"))
          hsplit(index*2,pts, {p1, p1_index}, pm)
        end),
        Task.async(fn ->
          pts = Keyword.get(:ets.lookup(:ch,
    String.to_atom("pts#{index}")), String.to_atom
    ("pts#{index}"))
          hsplit(index*2 + 1,pts, pm, {p2,
    p2_index})
```

```
        end)
      ])

      first ++ second
    end
  end
end
```

3.8 Ray Cast

3.8.1 The problem. Let R be a set of rays, each defined by a three-dimensional origin point, and a direction vector. Let T be a set of triangles, each defined by their vertices, which are three-dimensional points. The goal is to compute, for each ray, the first triangle that the ray would intersect had it been casted from its origin point, and return its index with respect to T (or -1 if the ray does not intersect any triangle).

3.8.2 Sequential implementation. We used the Möller–Trumbore algorithm [5] to compute the intersection point between a ray and a triangle. This algorithm is very efficient, and returns both a flag indicating if the ray intersects the triangle, and the exact point of intersection, if any. The implementation is as follows:

```
def ray_triangle_intersect(ray, triangle) do
  epsilon = 1.0e-8

  v0v1 = Vec3.sub(Enum.at(triangle.points, 1),
    Enum.at(triangle.points, 0))
  v0v2 = Vec3.sub(Enum.at(triangle.points, 2),
    Enum.at(triangle.points, 0))

  pvec = Vec3.cross_product(ray.to, v0v2)
  det = Vec3.dot_product(v0v1, pvec)

  if abs(det) < epsilon do
    {false, nil}
  else
    inv_det = 1 / det
    t_vec = Vec3.sub(ray.from, Enum.at(triangle.
    points, 0))
    u = Vec3.dot_product(t_vec, pvec) * inv_det
    if u < 0 or u > 1 do
      {false, nil}
    else
      q_vec = Vec3.cross_product(t_vec, v0v1)
      v = Vec3.dot_product(ray.to, q_vec) *
    inv_det

      if v < 0 or u + v > 1 do
        {false, nil}
      else
        dist = Vec3.dot_product(v0v2, q_vec) *
    inv_det

        {true, dist}
      end
    end
  end
end
```

Both the sequential and parallel implementations use this algorithm as a subroutine. The sequential implementation performs a nested loop to calculate every intersection point, and for each ray, takes the index of the triangle associated with the minimum distance:

```
def ray_cast(triangles, rays) do
  indexed_triangles = Enum.with_index(triangles)

  result = Enum.map(rays, fn ray ->
    Enum.map(indexed_triangles, fn ({triangle,
    index}) ->
      {ray_triangle_intersect(ray, triangle),
    index}
    end)
    |> Enum.filter(fn ({{intersects, _distance},
    _index}) -> intersects end)
    |> Enum.min_by(fn ({{_intersects, distance},
    _index}) -> distance end, fn -> -1 end)
    |> take_index
  end)

  result
end

defp take_index(-1) do
  -1
end

defp take_index({{_intersects, _distance}, index})
     do
  index
end
```

3.8.3 Parallel implementation. The parallel implementation uses *ets* to broadcast rays and triangles to workers, and each processes a subset of the rays in parallel. Then, at the end, results are combined:

```
def ray_cast(triangles, rays, p) do
  :ets.new(:rc, [:public, :named_table])
  indexed_triangles = Enum.with_index(triangles)
  :ets.insert(:rc, {:triangles, indexed_triangles
    })
  :ets.insert(:rc, {:rays, rays})

  size = ceil(length(rays) / (p-1))
  result = (0..p-1)
  |> Enum.map(fn idx -> (idx*size) end)
  |> Enum.map(fn start ->
    Task.async(fn ->
      tri = Keyword.get(:ets.lookup(:rc, :
    triangles), :triangles)
      Keyword.get(:ets.lookup(:rc, :rays), :rays)
      |> Enum.slice(start, size)
      |> Enum.map(fn ray ->
        Enum.map(tri, fn ({triangle, index}) ->
```

```
          {PBBS.Geometry.RayCast.Sequential.
    ray_triangle_intersect(ray, triangle), index}
        end)
        |> Enum.filter(fn ({{intersects, _distance
    }, _index}) -> intersects end)
        |> Enum.min_by(fn ({{_intersects, distance
    }, _index}) -> distance end, fn -> -1 end)
        |> take_index
      end)
    end)
  end)
  |> Task.await_many(:infinity)
  |> :lists.append

  :ets.delete(:rc)
  result
end
```

4 Experiments

We conducted a series of experiments to evaluate the parallel implementations against the sequential implementations, for each problem. The main goal is to assess the speedups obtainable by the parallel implementations, answering the following question:

- What speedup the parallel implementation achieves?

4.1 Experimental configuration

Software. The algorithms were implemented in version 1.14 of the *Elixir* language, compiled with Erlang/OTP 24. The operating system used was Ubuntu 20.04, 64 bits.

Hardware. The experiments were conducted in a 40 core machine, with maximum clock of 3600 MHz, and 32GB of RAM.

Benchmarks. For some problems, different inputs were used to exercise different communication patterns. For example, in the word count problem, strings with fewer unique words lead to less overall communication, which in turn affect speedups.

Methodology. Each input-implementation pair is warmed up for 2 seconds, and then executed for 60 seconds, with the aid of the third-party library *benchee*. Reported execution times are the average in this 60 seconds window.

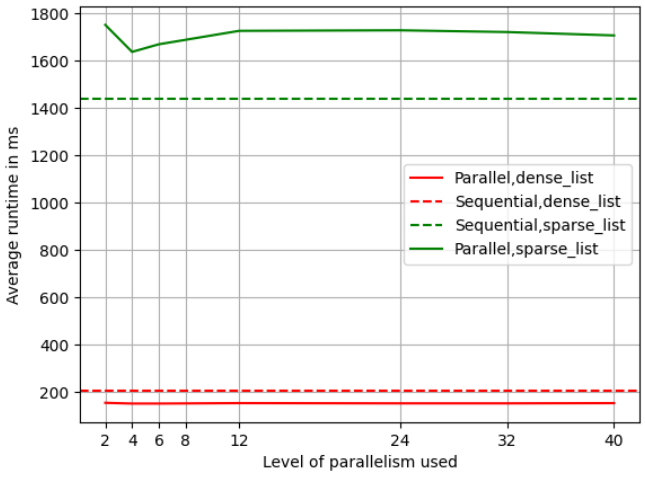
4.2 Experimental results

4.2.1 Histogram. For the histogram problem, two different inputs were used:

- *dense list*: a list with 1 million integers in the $[0, 1000]$ range
- *sparse list*: a list with 1 million integers in the $[0, 500000]$ range

These inputs are also used for the experiments of remove duplicates and both sorting algorithms.

Figure 1. Average Histogram execution time per level of parallelism



In figure 1, we show the average execution time of each input-implementation pair, ranging the parallelism level from 2 to 40.

It is clear that the input impacts the capability of the parallel implementation to outperform the sequential implementation. For the *dense list*, as less communication happens, the parallel implementation manages to achieve a 1.36 speedup over the sequential implementation. On the other hand, for the *sparse list*, communication overheads are larger, and the parallel implementation is slower than the sequential implementation (*speedup* of 0.88).

Also, we can note that the level of parallelism does not affect the parallel algorithm performance. That also can be attributed to communication overheads: as the problem intrinsic complexity is linear, most time is spent communicating, and most cores remain underutilized.

4.2.2 Remove Duplicates. The *dense_list* and *sparse_list* inputs were also used for the remove duplicates benchmarks. In figure 2, we show the execution times. By using the broadcasting strategy, the parallel version can achieve speedups for both the inputs, unlike in histogram. However, broadcasting comes with a trade-off: as the level of parallelism grows, each worker ends up taking a higher proportion of time receiving and dropping unused data, and that leads to a performance deterioration. It is also worth noting that the input also impacts the speedups: for the dense list, the parallel implementation outperforms the sequential implementation for more levels of parallelism, by a greater degree, and performance deteriorates slower. The best speedup obtained with the dense list is 1.51 at $p = 4$, and for the sparse list, the best speedup value is 1.31, also obtained at $p = 4$.

Figure 2. Average Remove Duplicates execution time per level of parallelism

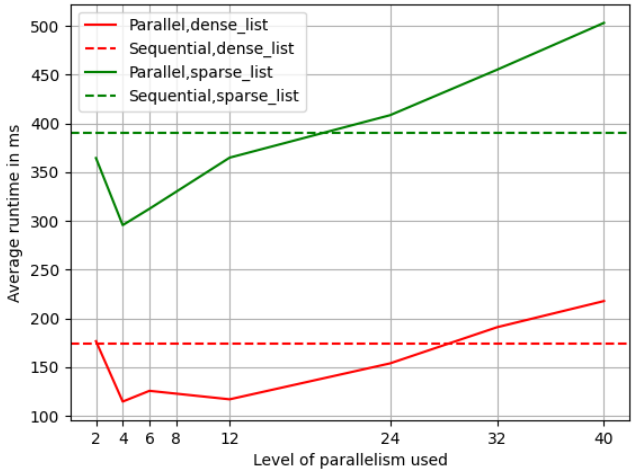
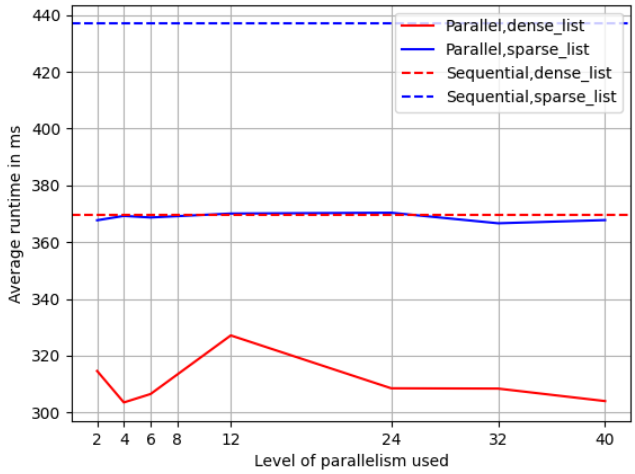


Figure 3. Average Integer Sort execution time per level of parallelism

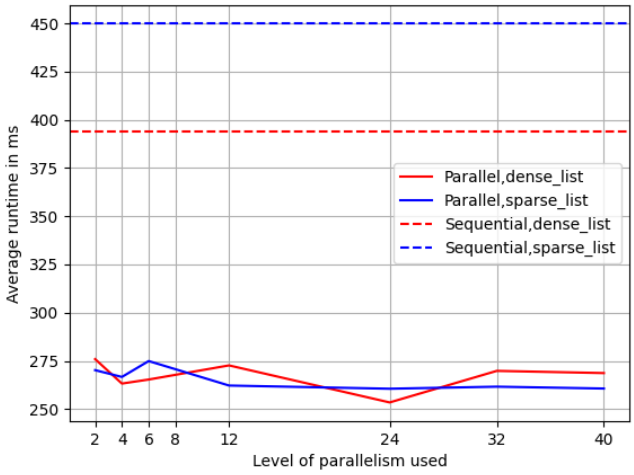


4.2.3 Integer Sort.

The same inputs used previously are used in integer sort. In figure 3, the execution times are shown.

In this case, better results were achieved through the use of the chunking strategy. As radix sort time complexity is dependent on the number of digits of the numbers, and by consequence more expensive than the previous problems, it is therefore expected that communication costs make up a smaller proportion of the overall execution time, leading to better speedups. For the dense list, the best speedup obtained was 1.22 at $p = 4$ and $p = 40$, and for the sparse list, 1.19 at $p = 32$. We also can see that the average runtime remains mostly constant irrespective of the parallelism level. We argue that despite the problem being more complex, the chunks are too small to make actors spend most of the time in computation, which is what would lead to better speedups as the parallelism level grows. With chunks sized 20000, each worker does $20000 \times log_{10}(500000) \approx 110000$ operations for the sparse list, which is likely not sufficient to outweigh communication costs.

Figure 4. Average Comparison Sort execution time per level of parallelism



4.2.4 Comparison Sort.

Comparison sort also used the same inputs. Figure 4 displays the results. Similar results were achieved in comparison to integer sort. As merge sort time complexity is somewhat higher than radix sort, each worker does more computation per chunk, so better speedups are achieved: 1.55 for the dense list and 1.72 for the sparse list, both at $p = 24$.

4.2.5 Word Count.

For the word count problem, two different inputs were used:

- *natural text*: 400K of natural text, extracted from *Project Gutenberg*
- *dense text*: 400K of synthetic text, composed of only 36 unique words

In figure 5, we show the experiments results for word count. In this case, performance improves as the parallelism level increase, despite word count being a linear time complexity problem. That has to do with the fact that the key-value structure that represents frequencies in a string is much smaller than the string itself, on average, so little communication happens from worker processes to the main process. For dense text, a speedup of 5.19 is achieved on $p = 12$, and for sparse text, a speedup of 3.03 is achieved also on $p = 12$.

4.2.6 Suffix Array.

Two inputs were used for the suffix array problem:

Figure 5. Average Word Count execution time per level of parallelism

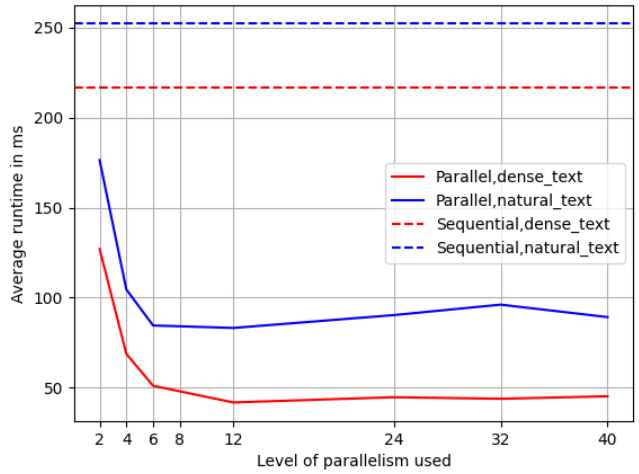
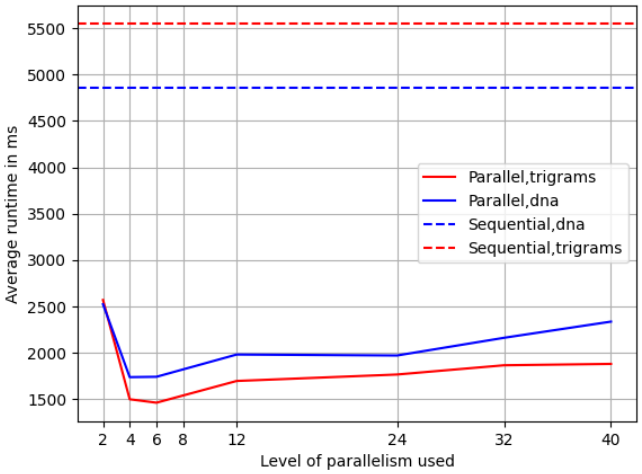


Figure 6. Average Suffix Array execution time per level of parallelism

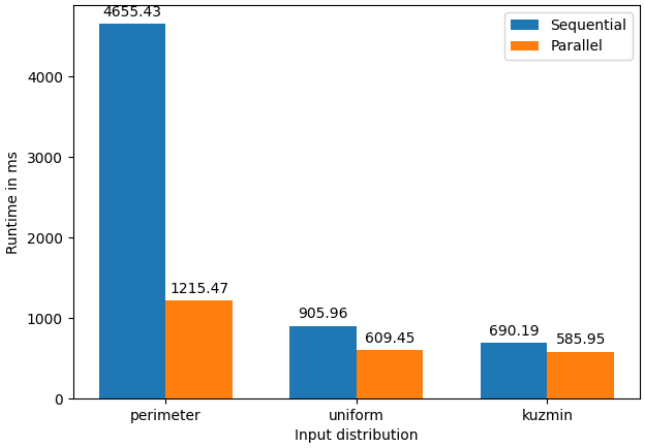


- *trigrams*: 1 million characters from a trigram distribution
- *dna*: 1 million characters from a dna sequence (only 5 unique characters: A, C, T, G and N)

Figure 6 shows the results per level of parallelism. Overall, larger speedups occur around $p = 4$ and $p = 6$, and then performance regressions happen, due to communication overheads (but even then, the parallel implementations remain much faster). For the *trigrams* input, a speedup of 3.79 is achieved at $p = 6$, and for the *dna* input, a speedup of 2.79 is achieved at $p = 4$.

4.2.7 Convex Hull. Three different inputs were used for the convex hull problem. All of them are composed by 300000 two-dimensional points, each following a different distribution:

Figure 7. Average Convex Hull execution time per level of parallelism



- *uniform* follows a uniform distribution in the unit-circle
- *perimeter* are points on the perimeter of the unit-circle
- *kuzmin* are points following the Kuzmin distribution

The results are in figure 7. As the level of parallelism is not parameterizable in the implemented parallel algorithm, we only show the execution times for $p = 40$, the number of cores in the machine. For the *perimeter* input, the achieved speedup was 3.83; for the *uniform* input, 1.49, and 1.18 for the *kuzmin* input. The larger speedup for the *perimeter* input is observed because this is a worst-case input for the *QuickHull* algorithm, so offloading work to other processes is very effective.

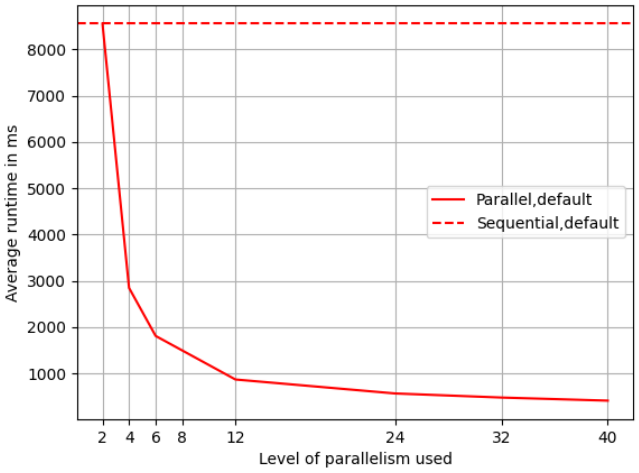
4.2.8 Ray Cast. Lastly, we show the results for the experiments on the ray cast problem. Only a single input (*default*) was used: a subset of 2000 triangles from the *Angel Triangles* dataset, and 2000 rays created using PBBS's own utilities. Figure 8 shows the results: we get continuously better runtimes as the level of parallelism increases.

As the implemented Ray Cast algorithm is the most expensive of the implemented sequential algorithms ($O(m * n + m\ log\ m)$ on the worst case, where m is the number of triangles and n is the number of rays), communication time is not the bottleneck of the workers (rather, computation time dominates), so better CPU usage and better speedups are achieved in this case: 20.67 on $p = 40$.

5 Conclusions

The main conclusion we can draw from this work is that communication is the main bottleneck for parallel algorithms in Elixir. While preventing the programmer from having

Figure 8. Average Ray Cast execution time per level of parallelism



to deal with explicit synchronization mechanisms, such as locks, the actor model requires that all communications happen through message passing, which is substantially more expensive than process-local computations.

The main factor that determines speedups are the proportion of time communicating versus the proportion of time doing useful computation. These two variables, on the other hand, depend on other factors:

- some problems require less communication, such as Ray Cast, whose result for each ray is a simple index
- some inputs cause the parallel algorithm to communicate less, such as *dense lists* on certain sequence problems
- some problems have higher intrinsic time complexity, so each worker gets to spend more time doing useful work, rather than receiving and sending messages

While interesting from an theoretical point of view and having a wide variety of practical applications, we can conclude that the actor model is not the best choice to parallelizing $O(n)$ and $O(n\ log\ n)$ algorithms. However, it starts to shine as problem complexity grows, or when dealing with I/O bound workflows.

References

[1] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab.

[2] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. 1996. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)* 22, 4 (1996), 469–483.

[3] Guy Edward Blelloch, Jeremy T. Fineman, Phillip Gibbons, Aapo Kyrola, Harsha Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. *Proc. ACM Symposium on Parallelism in Algorithms and Architecture (SPAA)* (2012).

[4] Paul Butcher. 2014. Seven Concurrency Models in Seven Weeks: When Threads Unravel. *Seven Concurrency Models in Seven Weeks* (2014), 45–46.

[5] Tomas Möller and Ben Trumbore. 2005. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*. 7–es.

[6] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. 2007. A Taxonomy of Suffix Array Construction Algorithms. *ACM Comput. Surv.* 39, 2 (jul 2007), 4–es. https://doi.org/10.1145/1242471.1242472

[7] Tim Rentsch. 1982. Object oriented programming. *ACM Sigplan Notices* 17, 9 (1982), 51–57.

[8] Anish Man Singh Shrestha, Martin C. Frith, and Paul Horton. 2014. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in Bioinformatics* 15, 2 (01 2014), 138–154. https://doi.org/10.1093/bib/bbt081 arXiv:https://academic.oup.com/bib/article-pdf/15/2/138/562332/bbt081.pdf