# COMS 4995 Project Report

# Parallelized Particle Swarm Optimization

Xijiao Li (xl2950)
Chen Chen (cc4351)

December 23, 2021

**Abstract**

In this report, we explore the magic of parallel computation in the pure functional language - Haskell. We present a parallelized version of Particle Swarm Optimization implementation in Haskell and examine the improvement on runtime performance. Its implementation is then manifested in a 10-dimensional Shekel Function with 10 minima. We have also tried different parallelization strategies and showed how each of them can improve the performance.

## 1 Introduction

Haskell[1] is an advanced, purely functional programming language invented in 1987 by a group of programming language researchers. Like other pure functional languages, Haskell is based on the lambda calculus, a formal mathematical system for expressing the notion of computation, invented by Turing's professor Alonzo Church. At a high-level, a Haskell program is just a function composed of many smaller functions.

Particle Swarm Optimization[2] (PSO) has been proven to be effective at solving complicated optimization problems, and has been successfully applied in a wide range of practical tasks. It is a nature-inspired computational method, in which a swarm of particles work together to practical problems, with each trying to improve his individual solution through the interaction with his neighbors. Since PSO does not use the gradient of the problem being optimized, it can be applied on non-smooth and non-convex functions as they are derivative-free and produce results independent of the initial model. Besides, there are very few hyperparameters, which are also simple to understand. For the same hyperparameters, PSO will work on a very wide variety of tasks, which makes it very powerful and flexible.

[1] `https://www.haskell.org/`
[2] `https://en.wikipedia.org/wiki/Particle_swarm_optimization`.

One caveat of particle swarm optimization algorithms is premature stagnation: the swarm may converge to a non-optimal solution. This can be caused by the fact that the step length is too large (so the particles ignorantly jump over the global optima), or the the number of particles is too small. However, if we set a tiny step length, the total number of iterations needed for the particles to converge will increase, leading to a longer running time.

## 2 Background

In the context of PSO, there are a number of particles moving through the search space in search of the best solution. Every particle position represents a potential solution and the goodness/fitness of that solution is measured by an objective function (the function being optimized).

Upon initialization, each particle is randomly assigned to a random position and with a random initial velocity. At each time step, every particle first updates its velocity and then the position:

$$
\begin{aligned}
V_{i,t+1} &= wV_{i,t} + c_1 r_1 (O_{i,t} - P_{i,t}) + c_2 r_2 (O_t - P_{i,t}) \\
P_{i,t+1} &= P_{i,t} + V_{i,t+1}
\end{aligned}
$$

where

- $V_{i,t} :=$ the velocity of particle $i$ at time $t$
- $P_{i,t} :=$ the position of particle $i$ at time $t$
- $O_{i,t} :=$ the position with min cost ever visited by particle $i$ at time $t$
- $O_t :=$ the position with min cost ever visited by any particle at time $t$
- $r_1, r_2, w, c_1, c_2$: different weight parameters for each term

One can see from the equation that the movement of the particles is governed by three factors: the *inertia weight* component ($V_{i,t}$), the *cognitive* component ($O_{i,t} - P_{i,t}$) and the *social* component ($O_t - P_{i,t}$). The inertia weight component allows a particle to maintain some momentum between iterations. The cognitive component allows the particle's movement to be influenced by its memory of good positions that it has found in earlier iterations. The social component will cause the good positions found by other particles of the swarm to influence the given particle's movement.

Overall the logic of PSO can be found in the pseudo code:

**Algorithm 1:** PSO algorithm

```
1  begin
2      Initialize N particles on the search space;
3      for time step t ← 0 to max_iter do
           // update all particles
4          for every particle i do
5              Update velocity to V_{i,t};
6              Make next movement to position to P_{i,t};
7              Calculate function value at its current position P_{i,t};
8              Find and update O_{i,t};
9          end
10         Find and update O_t;
11     end
12 end
```

## 3 Haskell Implementation

In this section, we will explain how we implement the parallelized PSO in Haskell.

### 3.1 Particle

Each particle is `Particle` type with some type parameters, which are the states of particle that we need to keep track of:

```haskell
data Particle = Particle
  { velocity :: [Double] -- ^ Velocity at time i
  , posCurr  :: [Double] -- ^ Position at time i
  , posBest  :: [Double] -- ^ Best position upon time i
  , valCurr  :: Double   -- ^ Current value at time i
  , valMin   :: Double   -- ^ Minimal value upon time i
  }
```

The function `update` updates the `Particle` passed in. Basically, it calculates the new velocity and new position for the particle, calculate the new value of the new position using the cost function, and update the parameters.

```haskell
update
  :: ([Double] -> Double) -- ^ Cost function
  -> [(Double, Double)]   -- ^ Cost function domain
  -> [Double]             -- ^ Current global optimal position
  -> Double               -- ^ Weight parameter
  -> Particle             -- ^ Particle to update
  -> Particle
```

```haskell
update fn domain posGOpt weight part = Particle { ... }
where
  ... -- calculate new parameters
```

### 3.2 PSO State

We implement the PSO algorithm using a state monad in order to keep track of the states for each iteration.

```haskell
data PSOState = PSOState
  { swarm        :: [Particle]           -- ^ Particle swarm
  , swarmSize    :: Int                  -- ^ Size of Particle swarm
  , costFn       :: [Double] -> Double   -- ^ Cost function
  , searchDomain :: [(Double, Double)]   -- ^ Cost function domain
  , currOptima   :: [Double]             -- ^ Current global optima
  , trueOptima   :: [Double]             -- ^ Real optima
  , timestamp    :: Int                  -- ^ Current timestamp
  , weight       :: Double               -- ^ Weight parameter
  , psoLog       :: [String]             -- ^ Trace log
  }
```

We use the function `runPSO` to run a whole session of PSO. The parameter `n` is the number of iterations to run. This function will recursively call itself until `n` hits 0; every time, it calls `step` to iterate one step (i.e. update each particle in the swarm, find the new global optima so far, and write it to the logs).

```haskell
runPSO :: Int -> State PSOState ()
runPSO 0 = return ()
runPSO n = do
  step
  runPSO n-1 -- go to next iteration

step :: State PSOState ()
step = do
... -- get sw, fn, domain, currOpt, w from state
let s'@(p:ps) = map (Particle.update fn domain currOpt w) s
    newOpt = Particle.posBest $ foldr minPart p ps
modify $ \st -> st { swarm = s'
                   , currOptima = newOpt
                   , timestamp = timestamp st + 1
                   , psoLog = show (timestamp st, newOpt) : psoLog st }
return ()
```

### 3.3 Cost Function

In order to test and experiment with the parallelized version of PSO implementation, we implement several cost functions in Haskell as well. Each function is in a separate module, which has implemented three things: `function`, `domain`, and `minima`. Below is an example of a 10-dimensional Shekel function with 30 local minima:

```
{- cVec and aVec are some constants for this function.
-}
function :: [Double] -> Double
function = (* (-1)) . inv . inner
  where
    inv x = sum \$ fmap (** (-1)) (zipWith (+) x cVec)
    inner p = map ((sum . map (** 2)) . zipWith (-) p) aVec

domain :: [(Double, Double)]
domain = replicate 10 (0, 10)

minima :: [Double]
minima = [ ... ]
```

### 3.4 Main

Inside `Main.hs`, we take in several parameters from the command line, initialize the initial PSO state, and kick off the iterations by calling `runPSO`. In order to assign each particle a random position and a random velocity when initializing the state, we use the `randomRIO` from `System.Random`.

```
ps <- replicateM s $ mapM randomRIO Shekel.domain -- initial positions
vs <- replicateM s $ mapM randomRIO Shekel.domain -- initial velocities
```

## 4 Parallelization

We have attempted several strategies to parallelize our PSO implementation, specifically the update step (lines 4 - 9 in the pseudocode of PSO Algorithm, with implementation details explained in §3). The general idea is to break up the computation at each update step into multiple chunks and create a spark for each chunk that will subsequently submit the computation to a thread pool to execute the computation in parallel.

### 4.1 parList rdeepseq

We started with a simple `parList rdeepseq` strategy, where we create 1 spark for each particle in each iteration to evaluate each particle to NFData form. We have

made the following code change to test this strategy:

```
let s'@(p:ps) = map (C.update currOpt fn domain w) s
                `using` parList rdeepseq

-- To allow our Particle type to be fully evaluated
instance NFData Particle where
  rnf (Particle v pC pB vC vMin) = rnf v `deepseq`
                                   rnf pC `deepseq`
                                   rnf pB `deepseq`
                                   rnf vC `deepseq`
                                   rnf vMin
```

We expect to see some speed up with `parList` strategy but also significant overhead caused by creating a lot of sparks. The reason is that the computation for updating one single particle inside one iteration does not cost that much time, so it is wasteful to create a separate spark for that.

This effect is actually more severe when we use a simple cost function (e.g., a 2-D Shekel function with only 3 minima), which takes even less time to compute. Therefore, we use the 10-D Shekel function with only 30 minima to make it a better candidate for parallelization.

### 4.2 Vanilla rpar rseq

The second strategy we have tried is the vanilla `rpar` `rseq` strategy, where we split the entire particle list into equal-sized chunks, and call `rpar` on each of the chunks. Then we call `rpar` on each chunk, waiting for the computation to finish and concatenate the result. We choose this strategy to avoid spark pool overflow problem encountered in §4.1 The strategy requires the following code change:

```
let s'@(p:ps) = runEval $ do
    let subSwarms = splitN chunksize s
    subSwarms' <- mapM (rparUpdate fn domain currOpt w) subSwarms
    mapM_ rseq subSwarms'
    return $ concat subSwarms'
    where
      rparUpdate fn domain currOpt w subSwarm =
        rpar $ deep $ map (P.update fn domain currOpt w) subSwarm
```

We expect this strategy to work reasonably well to distribute workload across cores

with low overhead, since we typically split it up into 1 - 16 chunks to avoid creating too many sparks.

### 4.3 parListChunk rdeepseq

The last strategy we have tried is `parListChunk rdeepseq`, which works similarly as the vanilla `rpar rseq` strategy in §4.3. It can help to keep the load well-balanced across threads.

```
let s'@(p:ps) = map (P.update fn domain currOpt w) s
                `using` parListChunk chunksize rdeepseq
```

We expect to see the best performance using this strategy after some fine-tuning on chunk size.

## 5 Evaluation

### 5.1 Performance Metric and Evaluation Setup

To evaluate the four strategies we have used to run PSO, we have written a Python script to run our PSO executable with the aforementioned strategies using 1, 2, 4, 6, 8, 10, 12, 14, 16 software threads (through -N). For the `rpar rseq` and the `parListChunk` strategies, we run 1, 2, 4, 8, 12, 16, 20, 25, 35, 45, 60, 75, 100, 150, 200, 250, 333, 400, 500 number of chunks for each thread configuration. All experiment configuration statistics are averaged over 5 iterations and run using 500 particles for 1000 iterations. All data points are collected on an M1 chip Mac Mini with 8 CPU threads.

The key performance metric is the runtime of the program, collected using Threadscope and the bash time utility program. We have also included the HEC traces generated from Threadscope to understand the load-balancing across cores. Aside from the runtime, we have also included qualitative analyses on performance stability, the distribution of workload across cores, and the distribution of spark creation events across time and core.

### 5.2 Sequential

The sequential implementation using -N1 averages $9.9422 \pm 0.039$ seconds. See Figure 1 for a Threadscope trace.

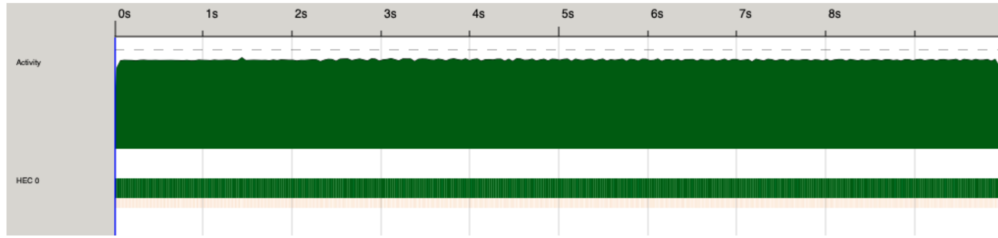


Figure 1: Sequential implementation Threadscope trace

| HEC | Total | Converted | Overflowed | Dud | GC'd | Fizzled |
|---|---|---|---|---|---|---|
| Total | 500000 | 8346 | 114806 | 0 | 0 | 16 |
| HEC 0 | 0 | 305 | 0 | 0 | 0 | 0 |
| HEC 1 | 0 | 286 | 0 | 0 | 0 | 0 |
| HEC 2 | 500000 | 1244 | 114806 | 0 | 0 | 16 |
| HEC 3 | 0 | 6511 | 0 | 0 | 0 | 0 |

Figure 2: Spark pool overflow with 500 particles using `parList` rdeepseq on 4 cores.

### 5.3 parList

The `parList rdeepseq` strategy provides $\sim$ 2.58x speedup compared with the sequential implementation when running with -N6 (best performance). The main problem with this strategy is that we create N sparks in each iteration, where N is the number of particles. It is necessary to have a large number of particles for PSO to fully explore the cost function domain to find the global minimum. As expected, we observe spark pool overflow with a large number of particles. See Figure 2 for details. We have also noticed more garbage collection and noticeable hiccups in the workload, suggesting significant overhead caused by the overwhelming number of sparks. Figure 2 also indicates that the load is not well-balanced, as HEC3 have the majority of the converted sparks.

See Figure 3 for performance analysis when running the `parList` strategy using different number of threads.

There are two observations. The first is that we see acceleration as we increase the number of threads running the workload up to 6 threads, which is as expected since we will have more threads to do the same amount of computation. Threadscope trace in 4 with workload distributed across cores also confirms our understanding. We also note that there is no performance increase, or even worsened performance, as we keep increasing the number of threads. We think the overhead from 1) kernel multiplexing more software threads than hardware threads (we only have 8); 2) distribute workload across many software threads are the main contributors to the worsened performance.

The second point is that the error bar is very noticeable, which is a result of occa-

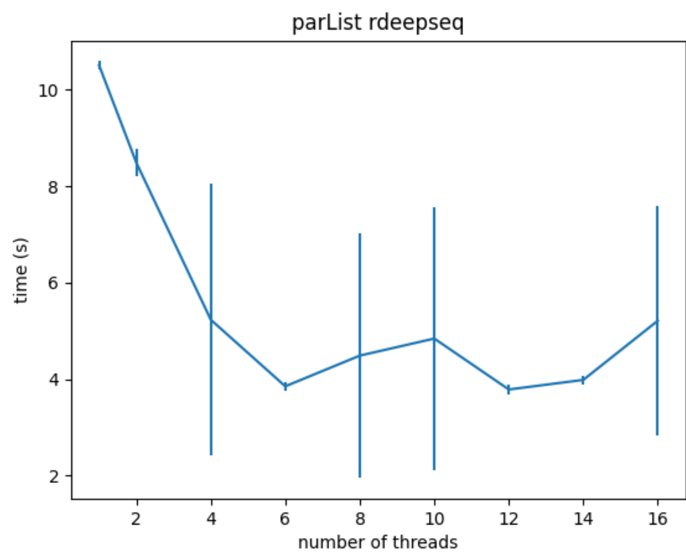


Figure 3: `parList` runtime over core: some speedup with increase of cores, significant fluctuations in performance.

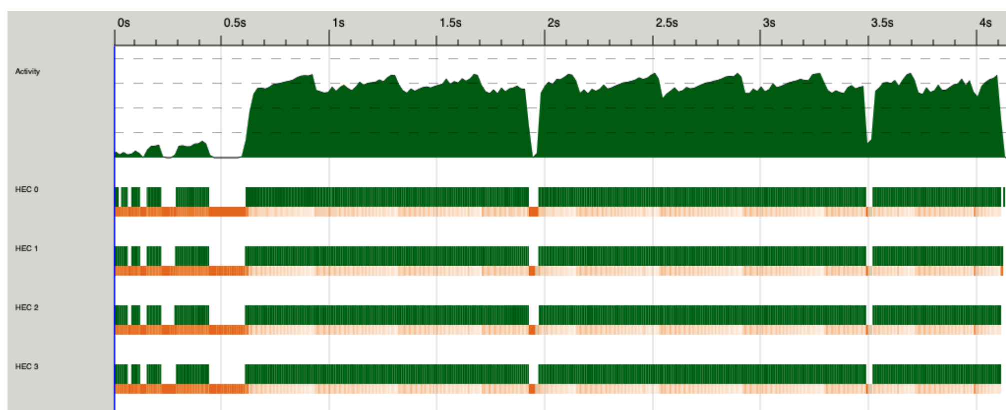


Figure 4: `parList` threadscope trace with -N4

| HEC | Total | Converted | Overflowed | Dud | GC'd | Fizzled |
|---|---|---|---|---|---|---|
| Total | 16000 | 14839 | 0 | 0 | 223 | 938 |
| HEC 0 | 8160 | 2699 | 0 | 0 | 126 | 123 |
| HEC 1 | 0 | 4610 | 0 | 0 | 0 | 384 |
| HEC 2 | 7136 | 2934 | 0 | 0 | 97 | 76 |
| HEC 3 | 704 | 4596 | 0 | 0 | 0 | 355 |

Figure 5: Better balanced workload, 16 chunks evaluated using rpar rseq on 4 cores.

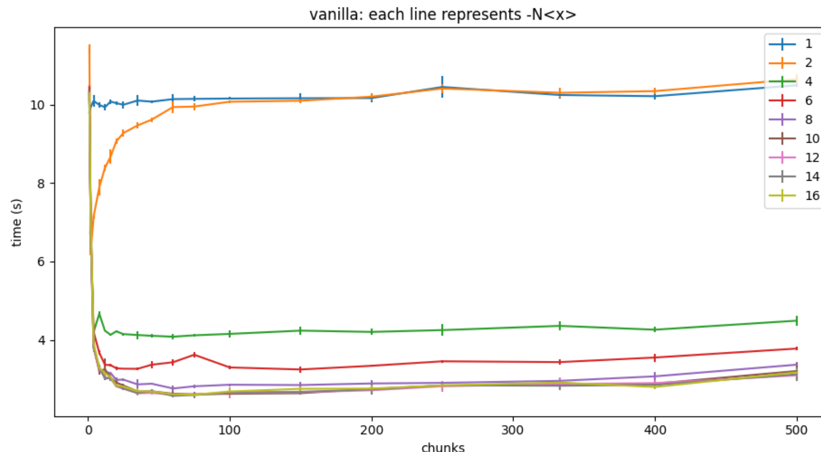


Figure 6: Vanilla `rpar rseq`: performance generally has minimal speed up after 16 chunks, plateaus at around 100 chunks with slight worse performance afterwards.

sional hiccup that causes a significant delay, usually in 1 out of the 5 iterations we run. The large standard deviations indicate that the `parList` strategy also has performance stability issues. The instability is likely a result of more garbage collection activities in the background as a result of the large overhead.

## 5.4 Vanilla rpar rseq

This strategy works reasonably well, providing ~3.18x speedup when compared against the baseline. We see from Figure 5 that it generates a relatively balanced workload across cores, which explains the increase in speedup. However, there remains a noticeable difference between the number of sparks converted by HEC 0 & 2 and HEC 1 & 3. We have also noticed a considerable number of sparks garbage collected and fizzled.

See Figure 6 for performance analysis when running the vanilla strategy using different number of threads and varying number of chunks. The first observation is

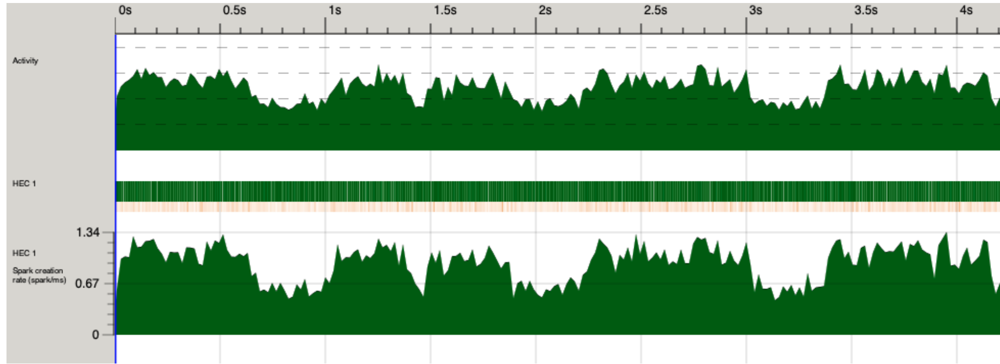


Figure 7: Vanilla strategy spark creation events are spread out to the entire runtime.

| HEC | Total | Converted | Overflowed | Dud | GC'd | Fizzled |
|---|---|---|---|---|---|---|
| Total | 16000 | 15989 | 0 | 0 | 3 | 8 |
| HEC 0 | 3408 | 4615 | 0 | 0 | 1 | 7 |
| HEC 1 | 1720 | 3463 | 0 | 0 | 0 | 0 |
| HEC 2 | 4577 | 3401 | 0 | 0 | 2 | 1 |
| HEC 3 | 6295 | 4510 | 0 | 0 | 0 | 0 |

Figure 8: parListChunk: good load-balancing across four cores with –chunk 16

that as we increase the number of cores up till 8 and the number of chunks up till 100 chunks, the greater the speedup. This is as expected. The second observation is that the standard deviation is significantly smaller than the data we have for the `parList` strategy in Figure 3. The small standard deviations indicate that spreading the spark creation across the entire duration helps to stabilize performance. Threadscope spark creation trace also confirms this observation in Figure (7. The third observation is that the performance worsens as the number of chunks goes beyond 100 due to the increased overhead brought by the increased number of threads.

### 5.5 parListChunk

The `parListChunk` strategy parallelizes the workload in a well-balanced way as well, as we could see from Figure 8 that 1) the spark conversion is well spread out across four cores; 2) the number of garbaged collected and fizzled sparks have decreased when compared with the vanilla strategy (Figure 5).

See Figure 9 for a performance analysis. Generally, the performance we get from using module `parListChunk` is very similar to that of the vanilla rpar rseq strategy. Overall, `parListChunk` is slightly better than the vanilla strategy in terms of speed up. See Figure 10 for a direct comparison.

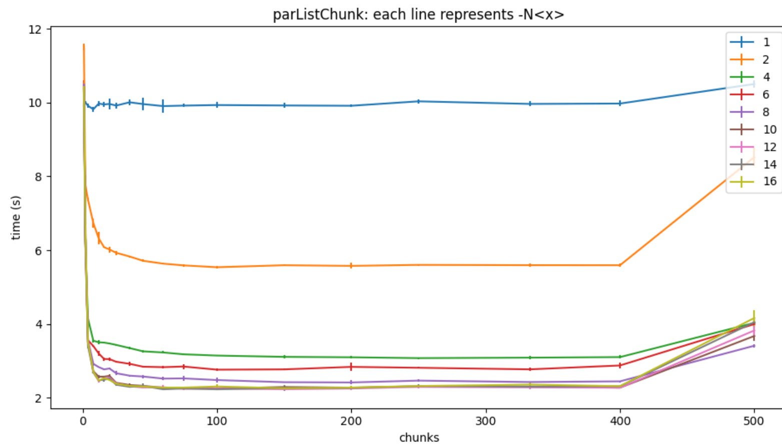


Figure 9: `parListChunk`: performance generally has minimal decrease after 16 chunks, plateaus at around 100 chunks with slight worse performance afterwards.

## 5.6 Strategies Comparison

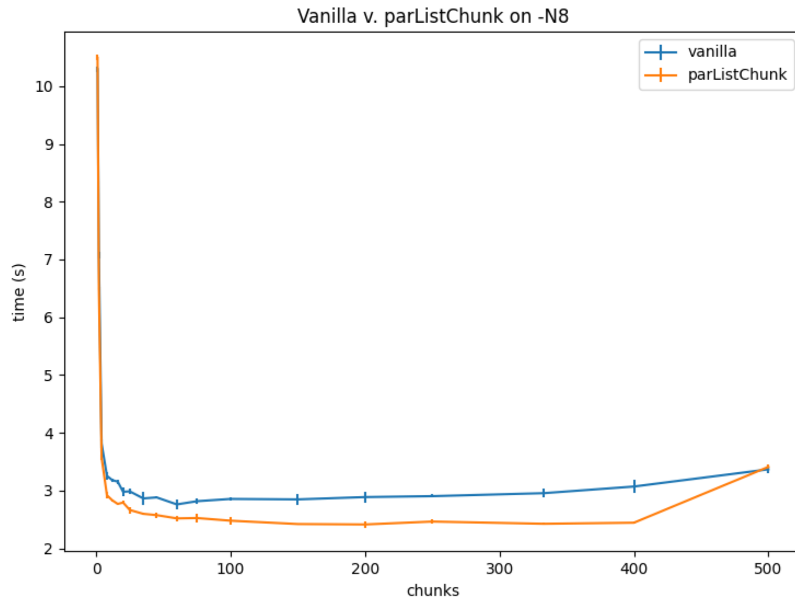


Figure 10: Comparing vanilla and `parListChunk` at -N8: `parListChunk` has better performance across configurations.

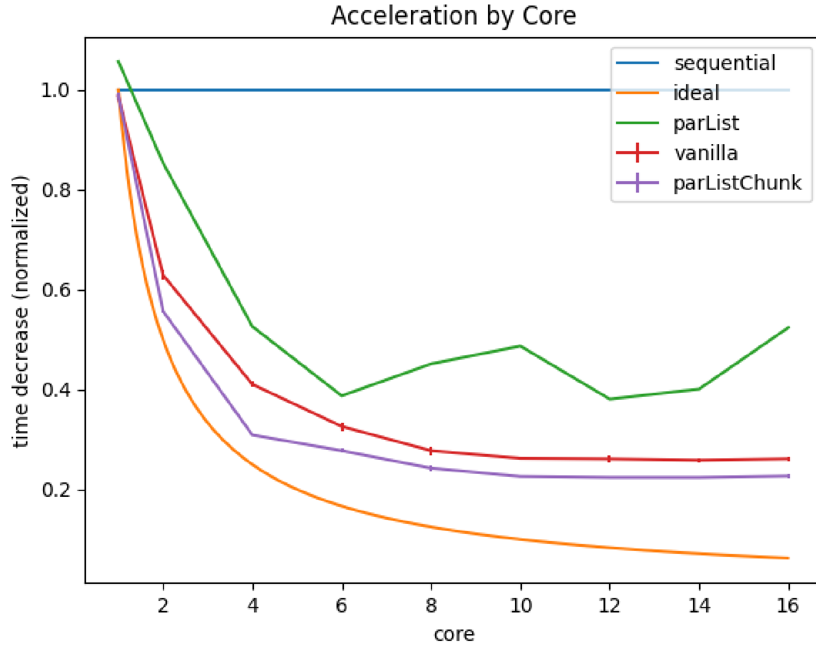


Figure 11: Best achieved performance for N = 500 particles, I = 1000 iteration

We conclude the evaluation section by comparing performance across strategies. We see that `parListChunk` has the overall best speed up, followed by vanilla strategy. As we increase the number of threads, we move further away from the ideal performance due to the sequential part of our implementation. It is also interesting that even though the experiment is run on a computer with only 8 hardware threads, we could still get performance increase as we increase from 8 software threads to 16 software threads. This indicates that the CPUs are idle part of the time when running with -N8 and there is room for multiplexing.

# 6 Conclusion

Haskell's robust runtime enables us to parallelize the PSO implementation by changing only several lines of the sequential version of PSO. From the last section, we can see that the performance of the algorithm greatly increases as expected. The `parListChunk` provided a very clean way to achieve the parallelization. Beside, the monad enables us to not worry about the side effects and put our focus on the algorithm itself. All of these contribute to our elegant and clean codes.

# References


[1] J. Kennedy and R. Eberhart, *Particle swarm optimization.* IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948, December 1995.

[2] M. Millonas, *Swarms, phase transitions and collective intelligence.* Artificial life III, C. Langton, Ed., pp. 417–445, Addison-Wesley, Reading, Mass, USA, 1994.

[3] I. M. Yassin, M. N. Taib, R. Adnan, M. K. M. Salleh, and M. K. Hamzah, *Effect of swarm size parameter on Binary Particle Swarm optimization-based NARX structure selection.* Proceedings of the IEEE Symposium on Industrial Electronics and Applications (ISIEA '12), pp. 219–223, Bandung City West Java, Indonesia, September 2012.

[4] S, Katare. and D.H, West, *Optimal complex networks spontaneously emerge when information transfer is maximized at least expense: A design perspective.* doi:10.1002/cplx.20119, 2006

[5] Arion de Campos, Aurora T.R. Pozo, Elias P. Duarte, *Parallel multi-swarm PSO strategies for solving many objective optimization problems.* Journal of Parallel and Distributed Computing, Volume 126, 2019, Pages 13-33, ISSN 0743-7315.

# A Main.hs

```
module Main where

import qualified Data.Map                          as M
import qualified Data.Set                          as S
import qualified Particle                          as P
import PSO

import System.Console.GetOpt
import Control.Monad
import Control.Monad.State
import Control.DeepSeq
import System.IO
import System.Exit
import System.Environment
import System.Random
import Data.List(unfoldr)
import qualified Shekel
import qualified Rastrigin


data Options = Options  { optVerbose       :: Bool
                        , optHelp          :: Bool
                        , optSwarmSize     :: Int
                        , optMaxIteration  :: Int
                        , optChunkNumber   :: Int
                        , optStrategy      :: Int
                        }

defaultOptions :: Options
defaultOptions = Options  { optVerbose       = False
                          , optHelp          = False
                          , optSwarmSize     = 500
                          , optMaxIteration  = 1000
                          , optChunkNumber   = 10
                          , optStrategy      = 2
                          }

options :: [ OptDescr (Options -> Options) ]
options =
```

```haskell
  [ Option "h" ["help"]
               (NoArg $ \opt -> opt { optHelp = True })
               "show help"
  , Option "v" ["verbose"]
               (NoArg $ \opt -> opt { optVerbose = True })
               "verbose"
  , Option "s" ["size"]
               (ReqArg (\str opt ->
                    opt { optSwarmSize = read str }) "SWARM_SIZE")
               "size of swarm"
  , Option "i" ["iteration"]
               (ReqArg (\str opt ->
                    opt { optMaxIteration = read str }) "MAX_ITERATION")
               "number of max iteration to run"
  , Option "c" ["chunk"]
               (ReqArg (\str opt ->
                    opt {optChunkNumber = read str}) "CHUNK_NUMBER")
               "number of chunks to run each iteration in"
  , Option "p" ["parallel"]
               (ReqArg (\str opt ->
                    opt {optStrategy = read str}) "PARALLEL_STRATEGY")
               "parallel strategies"
  ]

main :: IO ()
main = do
  args <- getArgs
  -- Parse options, getting a list of option actions
  let (actions, nonOptions, errors) = getOpt RequireOrder options args
      -- Here we thread startOptions through all supplied option actions
      opts = foldl (flip id) defaultOptions actions
      Options { optVerbose       = v
              , optHelp          = h
              , optSwarmSize     = s
              , optMaxIteration  = i
              , optChunkNumber   = c
              , optStrategy      = p } = opts

  if h || length errors /= 0 || p < 0 || p > 3 then usage
    else putStrLn $ "Running with " ++ (show s) ++ " particles, "
                 ++ (show i) ++ " iterations, " ++ (show c) ++ " chunks " ++
                 ", strategy " ++ case p of
```

```
                                                    0 -> "sequential..."
                                                    1 -> "vanilla..."
                                                    2 -> "parList rdeepseq..."
                                                    _ -> "parListChunk rdeepseq..."

  ps <- replicateM s $ mapM randomRIO Shekel.domain
  vs <- replicateM s $ mapM randomRIO Shekel.domain
  let sw = P.initSwarm Shekel.function ps vs
      chunksz = (s + c - 1) `div` c
      (_, b) = runState (runPSO p chunksz i) PSOState
                 { swarm = sw
                 , swarmSize = s
                 , costFn = Shekel.function
                 , searchDomain = Shekel.domain
                 , trueOptima = Shekel.minima
                 , currOptima = map (const 0.0) Shekel.minima
                 , timestamp = 0
                 , weight = 0.5
                 , psoLog = [] }
  putStrLn $ unlines (take 1 (psoLog b))

usage :: IO ()
usage = do pn <- getProgName
           die $ "Usage: " ++ pn ++
             " [-hv] [-n <number particles>] [-i <number iterations>]" ++
             " [-c <number of chunks>]" ++
             " [-p <parallel strategy: " ++
             " {0:sequential, 1:vanilla, 2:parList, 3:parListChunk}>]"
```

# B Particle.hs

---

```
module Particle
  ( Particle(..)
  , initSwarm
  , update
  ) where
import Control.DeepSeq

data Particle = Particle
  { velocity :: [Double] -- ^ Velocity at time i
  , posCurr  :: [Double] -- ^ Position at time i
```

```haskell
  , posBest  :: [Double] -- ^ Best position upon time i
  , valCurr  :: Double   -- ^ Current value at time i
  , valMin   :: Double   -- ^ Minival value upon time i
  } deriving (Show)

instance NFData Particle where
  rnf (Particle v pC pB vC vMin) = rnf v `deepseq`
                                   rnf pC `deepseq`
                                   rnf pB `deepseq`
                                   rnf vC `deepseq`
                                   rnf vMin

initSwarm :: ([Double] -> Double) -> [[Double]] -> [[Double]] -> [Particle]
initSwarm fn = zipWith (\ p v ->
  Particle { velocity = v
           , posCurr = p
           , posBest = p
           , valCurr = fn p
           , valMin  = fn p })



update
  :: ([Double] -> Double) -- ^ Cost function
  -> [(Double, Double)]   -- ^ Cost function domain
  -> [Double]             -- ^ Current global optimal position
  -> Double               -- ^ Weight parameter
  -> Particle             -- ^ Particle to update
  -> Particle
update fn domain posGOpt weight part = Particle { velocity = v'
                                                , posCurr  = p''
                                                , posBest  = pb'
                                                , valCurr  = val
                                                , valMin   = min' }
  where
    [r1, r2] = [0.5, 0.5]
    c1 = 4.1 * weight
    c2 = 4.1 * (1-weight)
    d1 = map (*(c1*r1)) $ zipWith (-) (posBest part) (posCurr part) --cognitive
    d2 = map (*(c2*r2)) $ zipWith (-) posGOpt (posCurr part) --social
    wV = velocity part --inertia
    v' = zipWith3 (\x y z -> x + y + z) wV d1 d2
```

```haskell
    p' = zipWith (+) v' $ posCurr part
    p'' = zipWith (\xi (lower, upper) -> min (max xi lower) upper) p' domain
    val = fn p''
    pb' = if val < valMin part then p'' else posBest part
    min' = if val < valMin part then val else valMin part
```

# C PSO.hs

---

```haskell
module PSO
  ( PSOState(..)
  , runPSO
  ) where

import Control.Monad.State
import Control.Parallel.Strategies
import Control.DeepSeq (deepseq)
import Particle as P

data PSOState = PSOState
  { swarm        :: [Particle]           -- ^ Particle swarm
  , swarmSize    :: Int                  -- ^ Size of Particle swarm
  , costFn       :: [Double] -> Double   -- ^ Cost function
  , searchDomain :: [(Double, Double)]   -- ^ Cost function domain
  , currOptima   :: [Double]             -- ^ Current global optima
  , trueOptima   :: [Double]             -- ^ Real optima
  , timestamp    :: Int                  -- ^ Current timestamp
  , weight       :: Double               -- ^ Weight parameter
  , psoLog       :: [String]             -- ^ Trace log
  }

{- `runPSO strat chunksize n` runs the `step` loop using the given strategy
   @strat@ and chunk size @chunksize@ for @n@ iterations.
-}
runPSO :: Int -> Int -> Int -> State PSOState ()
runPSO _ _ 0 = return ()
runPSO strat chunksize n = do
  step strat chunksize
  -- go to next iteration
  runPSO strat chunksize $ n-1

{- `step strat chunksize` runs one `step`. @strat@ is the index of the
```

```haskell
   parallelization strategy to use and @chunksize@ is the number of particles
   that will be updated in parallel.
-}
step :: Int -> Int -> State PSOState ()
step strat chunksize = do
  s <- gets swarm
  fn <- gets costFn
  domain <- gets searchDomain
  currOpt <- gets currOptima
  w <- gets weight
  let s'@(p:ps) =
        -- three ways to parallelize
        case strat of
          {- vanilla split w/ rpar + rseq -}
          1 -> runEval $ do
            let subSwarms = splitN chunksize s
            subSwarms' <- mapM (rparUpdate fn domain currOpt w) subSwarms
            mapM_ rseq subSwarms'
            return $ concat subSwarms'
          {- parList -}
          2 -> map (P.update fn domain currOpt w) s
               `using` parList rdeepseq
          {- parListChunk -}
          3 -> map (P.update fn domain currOpt w) s
               `using` parListChunk chunksize rdeepseq
          {- sequential -}
          _ -> map (P.update fn domain currOpt w) s
      -- after we update all the particles, we find the new global optima
      newOpt = P.posBest $ foldr minParticle p ps
  -- update the PSO state
  modify $ \st -> st { swarm = s'
                     , currOptima = newOpt
                     , timestamp = timestamp st + 1
                     , psoLog = show (timestamp st, newOpt) : psoLog st }
  return ()
    where
      rparUpdate fn domain currOpt w subSwarm =
        rpar $ deep $ map (P.update fn domain currOpt w) subSwarm

-- `minParticle p1 p2` compares two Particles and return the one with smaller valMin
minParticle :: Particle -> Particle -> Particle
minParticle p1 p2
```

```haskell
  | P.valMin p1 < P.valMin p2 = p1
  | otherwise = p2

deep :: NFData a => a -> a
deep a = deepseq a a

splitN :: Int -> [Particle] -> [[Particle]]
splitN chunksize ps
  | chunksize >= length ps = [ps]
  | otherwise = ps1:splitN chunksize ps2
      where (ps1, ps2) = splitAt chunksize ps
```

## D Shekel.hs

---

```haskell
{- | Shekel function

See <https://en.wikipedia.org/wiki/Shekel_function>

Global minimum: shekel(0.5,0.5) = -594.960255
Bounds: -1 <= xi <= 1
-}
module Shekel
  ( function
  , domain
  , minima ) where

-- 30 local maxima.
aVec :: [[Double]]
aVec =
  [ [9.681, 0.667, 4.783, 9.095, 3.517, 9.325, 6.544, 0.211, 5.122, 2.020],
    [9.400, 2.041, 3.788, 7.931, 2.882, 2.672, 3.568, 1.284, 7.033, 7.374],
    [8.025, 9.152, 5.114, 7.621, 4.564, 4.711, 2.996, 6.126, 0.734, 4.982],
    [2.196, 0.415, 5.649, 6.979, 9.510, 9.166, 6.304, 6.054, 9.377, 1.426],
    [8.074, 8.777, 3.467, 1.863, 6.708, 6.349, 4.534, 0.276, 7.633, 1.567],
    [7.650, 5.658, 0.720, 2.764, 3.278, 5.283, 7.474, 6.274, 1.409, 8.208],
    [1.256, 3.605, 8.623, 6.905, 0.584, 8.133, 6.071, 6.888, 4.187, 5.448],
    [8.314, 2.261, 4.224, 1.781, 4.124, 0.932, 8.129, 8.658, 1.208, 5.762],
    [0.226, 8.858, 1.420, 0.945, 1.622, 4.698, 6.228, 9.096, 0.972, 7.637],
    [7.305, 2.228, 1.242, 5.928, 9.133, 1.826, 4.060, 5.204, 8.713, 8.247],
    [0.652, 7.027, 0.508, 4.876, 8.807, 4.632, 5.808, 6.937, 3.291, 7.016],
    [2.699, 3.516, 5.874, 4.119, 4.461, 7.496, 8.817, 0.690, 6.593, 9.789],
```

```
    [8.327, 3.897, 2.017, 9.570, 9.825, 1.150, 1.395, 3.885, 6.354, 0.109],
    [2.132, 7.006, 7.136, 2.641, 1.882, 5.943, 7.273, 7.691, 2.880, 0.564],
    [4.707, 5.579, 4.080, 0.581, 9.698, 8.542, 8.077, 8.515, 9.231, 4.670],
    [8.304, 7.559, 8.567, 0.322, 7.128, 8.392, 1.472, 8.524, 2.277, 7.826],
    [8.632, 4.409, 4.832, 5.768, 7.050, 6.715, 1.711, 4.323, 4.405, 4.591],
    [4.887, 9.112, 0.170, 8.967, 9.693, 9.867, 7.508, 7.770, 8.382, 6.740],
    [2.440, 6.686, 4.299, 1.007, 7.008, 1.427, 9.398, 8.480, 9.950, 1.675],
    [6.306, 8.583, 6.084, 1.138, 4.350, 3.134, 7.853, 6.061, 7.457, 2.258],
    [0.652, 2.343, 1.370, 0.821, 1.310, 1.063, 0.689, 8.819, 8.833, 9.070],
    [5.558, 1.272, 5.756, 9.857, 2.279, 2.764, 1.284, 1.677, 1.244, 1.234],
    [3.352, 7.549, 9.817, 9.437, 8.687, 4.167, 2.570, 6.540, 0.228, 0.027],
    [8.798, 0.880, 2.370, 0.168, 1.701, 3.680, 1.231, 2.390, 2.499, 0.064],
    [1.460, 8.057, 1.336, 7.217, 7.914, 3.615, 9.981, 9.198, 5.292, 1.224],
    [0.432, 8.645, 8.774, 0.249, 8.081, 7.461, 4.416, 0.652, 4.002, 4.644],
    [0.679, 2.800, 5.523, 3.049, 2.968, 7.225, 6.730, 4.199, 9.614, 9.229],
    [4.263, 1.074, 7.286, 5.599, 8.291, 5.200, 9.214, 8.272, 4.398, 4.506],
    [9.496, 4.830, 3.150, 8.270, 5.079, 1.231, 5.731, 9.494, 1.883, 9.732],
    [4.138, 2.562, 2.532, 9.661, 5.611, 5.500, 6.886, 2.341, 9.699, 6.500]
  ]

cVec :: [Double]
cVec =
  [ 0.806,
    0.517,
    0.1,
    0.908,
    0.965,
    0.669,
    0.524,
    0.902,
    0.531,
    0.876,
    0.462,
    0.491,
    0.463,
    0.714,
    0.352,
    0.869,
    0.813,
    0.811,
    0.828,
    0.964,
```

```
    0.789,
    0.360,
    0.369,
    0.992,
    0.332,
    0.817,
    0.632,
    0.883,
    0.608,
    0.326
  ]

function :: [Double] -> Double
function = (* (-1)) . inv . inner
  where
    inv x = sum $ fmap (** (-1)) (zipWith (+) x cVec)
    inner p = map ((sum . map (** 2)) . zipWith (-) p) aVec

domain :: [(Double, Double)]
domain = replicate 10 (0, 10)

minima :: [Double]
minima = [8.025, 9.152, 5.114, 7.621, 4.564, 4.711, 2.996, 6.126, 0.734, 4.982]
```

## E Rastrigin.hs

* This is another objective function that can be used to test the PPSO.

---

```
module Rastrigin
  ( function
  , domain ) where

{- | Rastrigin's function

See <https://en.wikipedia.org/wiki/Rastrigin_function>

Global minimum: rastrigin(0, 0) = 0
Bounds: -5.12 <= xi <= 5.12
-}
function :: [Double] -> Double
function p = 10 * n + sum inner
```

```haskell
  where
    n = fromIntegral $ length p
    inner = map (\x -> x**2 - 10 * cos (2 * pi * x)) p

domain :: [(Double, Double)]
domain = [(-5.12, 5.12), (-5.12, 5.12)]
```