COMS 4995 Project Report
Parallel Particle Swarm Optimization

Vincent Hugh Geiger(vhg2106), Siyi Hong(sh4325)

December 20, 2023

1 Introduction

1.1 Topic Introduction

Particle Swarm Optimization (PSO) is a computational method that originates from the simulation of social behavior patterns observed in nature, particularly those of fish schooling and bird flocking. Conceptualized by Kennedy and Eberhart in 1995, PSO is a metaheuristic algorithm, meaning it makes few assumptions about the problem being optimized and can search very large spaces of candidate solutions. Unlike other optimization algorithms that require gradient information, PSO optimizes a problem by iteratively improving a candidate solution with regard to a given measure of quality, or fitness function.
PSO works by initializing a group of random solutions, known as particles, and then moving these particles around in the search space according to simple mathematical formulae. Each particle's movement is influenced by its local best known position and the global best known positions in the search space, which are updated as better positions are found by the particles. This process is repeated until a satisfactory solution is discovered or a predefined criterion is met. PSO is widely used in various fields such as engineering, economics, data mining, and artificial intelligence due to its simplicity, efficiency, and ability to converge to a good solution with relatively few tunable parameters.

1.2 Motivation

The motivation for parallelizing the Particle Swarm Optimization algorithm stems from the ever-growing complexity and size of optimization problems in modern computational tasks. As problem sizes and complexities increase, the computational time of traditional, sequential PSO algorithms becomes a bottleneck, often rendering them impractical for large-scale applications. Parallel computing offers a solution to this challenge by distributing the computational workload across multiple processing elements, thereby significantly reducing execution times.
In PSO, each particle's position and velocity can be updated relatively independently, making it a prime candidate for parallelization. By leveraging modern multi-core and distributed computing environments, parallelPSO can handle larger problem sizes more efficiently, and provide solutions in a fraction of the time required by its sequential counterpart.

1.3 Objective and Structure

We aim to compare different parallelization methods, including ‘parList rdeepseq’ and ‘parListChunk rdeepseq’, and seek to understand how each strategy affects the overall performance and scalability of the PSO algorithm, particularly in terms of execution speed and resource utilization. Additionally, we aim to investigate the impact of various experimental parameters on the performance of parallelPSO. This includes examining the influence of parameters like swarm size, number of iterations, and chunk size on the optimization process. The report is structured to provide a comprehensive analysis of the parallelization strategies applied to Particle Swarm Optimization (PSO). It begins with an introduction to PSO and its significance, followed by a detailed background on PSO and parallel computing. The core of the report focuses on the implementation of PSO in Haskell, exploring various parallelization strategies, and an extensive evaluation of their performance. The report concludes with a summary of findings and potential directions for future research. Appendices with code listings and additional data are included for reference.

2 Background

2.1 PSO Algorithm Basics

The core of PSO lies in its method of updating each particle's velocity, which in turn guides the search process. The velocity of a particle is updated each iteration using the following formula:

$$v_i(t + 1) = wv_i(t) + c_1r_1(pbest_i - x_i(t)) + c_2r_2(gbest_i - x_i(t))$$

In this equation:

$v_i(t)$ is the current velocity of the i-th particle;

w represents the inertia weight, a parameter that controls the impact of the particle's previous velocity on its current one;

c_1 and c_2 are coefficients for cognitive and social components, respectively, balancing the influence of the particle's own experience and the swarm's collective experience;

r_1 and r_2 are vectors of random numbers, introducing stochasticity into the process. We settled on purely using 0.5 for both values.

$pbest_i$ denotes the best position found by the i-th particle so far (personal best).

$gbest_i$ is the best position found by any particle in the swarm (global best).

Following the velocity update, each particle in the swarm updates its position. Since each particle in the swarm updates its state (velocity and position) independently of others, this presents an ideal scenario for parallel computation. By updating multiple particles concurrently, parallelPSO can significantly enhance the exploration of the search space, leading to faster convergence towards optimal solutions, especially in problems with high-dimensional and complex search spaces.

2.2 Parallel Programming in Haskell

Haskell offers robust support for parallel programming, allowing developers to leverage multiple processors to speed up program execution. There are two main avenues to achieve parallelism in Haskell:
Pure Parallelism (Control.Parallel): This approach is suitable for speeding up pure computations. It guarantees determinism (the same result every time) and eliminates issues like race conditions and deadlocks. The par and pseq functions from the parallel library are fundamental tools in this category.
Concurrency (Control.Concurrent): This method is used for parallelizing IO operations. It involves multiple threads of control executing simultaneously, with IO operations from multiple threads interleaved non-deterministically.

Two notable strategies from the Control.Parallel.Strategies module in Haskell that are particularly relevant for parallelizing computations over lists are parList and parListChunk:
parList: This strategy is used to evaluate each element of a list in parallel. It applies a given strategy to each element of the list. For example, parList rseq would evaluate each element of the list to weak head normal form in parallel.
parListChunk: This strategy divides a list into chunks of specified size and then applies a strategy to each chunk in parallel. It is useful for controlling the granularity of parallelism and can be more efficient than parList for large lists, as it reduces the overhead associated with parallel task management. An example use case would be parListChunk chunkSize rdeepseq, where chunkSize is the size of each chunk, and rdeepseq ensures that each element in a chunk is fully evaluated.

3 Implementation:

3.1 Defining Data Types and States

The total state used to implement our qualities is a Swarm. It deals with qualities we want to call in a global context, and stores our list of particles. The data types of these particles are defined as follows:

```
data Particle =  Particle {position :: [Double],
                           velocity :: [Double],
                                      bestPos :: [Double],
                                      bestFitness :: Double} deriving (Show)
```

The type defines a position vector, velocity vector, and stores both bestPos (best position) and bestFitness (best fitness value). When we update our particle we utilize these vectors, and therefore it is necessary to define these in our type. Although it is possible to define only bestPos and simply calculate the bestFitness when needed, this leads to an overlap in calculations, requiring the calculation of bestFitness multiple times within every iteration. This increases runtime and is simply inefficient. The total Swarm State is defined as follows:

```
data SwarmState = SwarmState
                   {particles :: [Particle],
                    size :: Int,
                    maxIters :: Int,
                    globalBest :: [Double],
                    inertia :: Double,
                    cognition :: Double,
                    social :: Double,
                    posDims :: Int,
                    veloDims :: Int,
                    chunk :: Int,
                    choice :: Int,
                    objFunc :: ([Double] -> Double)}
```

The state will store our global list of particles, as well as various parameters required for the algorithm as well as our parallel implementation of it. We also define choice and objFunc within the state such that we can deal with a multitude of various objective functions and parallelization strategies.

3.2 Defining functions

We must first initialize our particles and swarm:

```
particleInit :: [Double] -> [Double] -> ([Double] -> Double) -> Particle
particleInit initPosition initVelocity objFunc' =
    Particle {position = initPosition, velocity = initVelocity,
              bestPos = initPosition, bestFitness = objFunc'
initPosition}
```

Our particleInit function takes as input an initial position and velocity vector (in the form of a list). These initial positions will be determined using random in our Main file. We then simply store this initial position as bestPos, as it is the only position the particle has ever explored. The objective function inputted is then applied to the initial position, as it must be the best fitness. Our swarm initialization deals with far more parameters. We define this later in our Main. Initial values are defined in our Main file as either hard-coded, allowing for change only though alteration of the program itself, or left up to user input. These initial values are simply defined as makes sense. The two initial values of note are the objective function and our initial list of particles. We then simply check which objective function the user inputted, and define the functions accordingly. The list of initial particles is determined by using map to particleInit for each of our initialVals, which is a list of tuples containing initial position and velocity vectors. These are determined using random in Main. During the runtime of our program, we must update our parameters based on our global parameters. To do so we run various updates on a particle, all accumulated using updateParticle:

```
-- combines both updateVelo and pos to update the particle
updateParticle :: Particle -> [Double] -> Double -> Double -> Double
-> ([Double] -> Double) -> Particle
updateParticle particle globalBest' inertia' cognitive' social'
objFunc' =
    updatePB (updatePos (updateVelo particle globalBest' inertia'
cognitive' social')) objFunc'
```

We simply update personal best using updatePB as one might expect. We compare our new position and its associated fitness value and return a particle with the better of the two. updateVelo simply computes our new velocity vector using zipWith4 and our defined formula for all relevant vectors and returns the modified particle. For updatePos, which updates the position of the particle, we want to update our particle's position using simply our defined formula. There are complications, however, as very commonly the particle can get stuck on the bounds. Despite there being how many iterations left over, the particle would simply cease to explore any particles on such an axis. To remedy this, we implemented bouncing.
If the particle's new position determined by our formula is within the bounds, we simply return our updated particle. If above the higher bound, or below the lower bound we simply set the position in that axis to double the respective bound minus the new position. Depending on how this occurs, however, the particle's velocity would remain unchanged, and thus would keep ramming into the wall over and over again. Thus, we also negate the velocity to ensure new particles will be explored.

```
updatePos :: Particle -> Particle
updatePos particle =
    particle {position = zipWith f' pos velo}
        where pos  = position particle
              velo = velocity particle
```

```
                f' p v  = p + 0.5*v
```

We update our velocity using zipWith alongside our defined formula. We also defined our various objective functions within SwarmOps.hs. Each function takes in a [Double] (our position vector) and outputs a Double. The functions we implemented are the sphere function, a quartic function, Rosenbrock's banana function, Ackley's function, and Rastrigin's objective function.

```
objectiveFunction:: [Double] -> Double
-- function definition
```

We also define an instance NFData Particle to later ensure that our parallelization will evaluate the particle in the way we desire.

3.3 entire program

The entire program works by iterating and updating the swarm until the defined max number of iterations is reached. The function iterateSwarm implements this. It takes in nothing and returns a SwarmState. Relevant values and items are obtained by utilizing the Swarm state. We then iterate through our particles using updateParticles, simply feeding in the necessary inputs. This is also where we choose which parallel strategy we will use.

```
iterateSwarm :: State SwarmState ()
iterateSwarm = do
    swarm <- get
    let -- define relevant values for use
        ...
        m  = fromIntegral(maxIters swarm)
        i' = i * ((m-1)/m)
        upParts =
           case (choice swarm) of
      -- use various strategies to update all of the particles and
modify the state of the swarm after determining the new global best
```

By updating our swarm's inertia in a dynamic way, we can aid particles. The inertia begins at 1, and as time goes on, reduces in value. In doing so, the particles are less likely to get stuck at local minima, and instead escape from them early on. Due to the dynamic nature of the inertia, however, later on a low inertia helps to ensure that a more accurate global minimum is discovered.

To begin the iterations we utilize our main function. We then initialize our swarm and feed it into our helper function, runPSOHelper, returning the output of that function. The helper calls itself with the updated swarm, reducing a stored number n, until eventually reaching 0, and simply returning.

```haskell
-- Helper function for recursive PSO iterations
runPSOHelper :: Int -> State SwarmState ()
runPSOHelper 0 = return ()
runPSOHelper n = do
    iterateSwarm -- Update the swarm
    runPSOHelper (n - 1) -- Continue the iterations
```

Our main function defines all of our initial values, printing out user-friendly messages. After ensuring that the correct number of arguments is inputted (an integer for choosing the method of parallelization, and another integer for choosing an objective function), our swarm is initialized. We utilize randomRIO when selecting the bounds to ensure some level stochasticity

```haskell
main :: IO ()
main = do
    args <- getArgs
    progName <- getProgName
    -- print messages
    case args of
        [parChoice, optEx] -> do
            let -- initialize values
            -- generate random position and velocity vectors
            -- read in which function has been chosen
               particles' = map (\(initP, initV) -> particleInit
initP initV objFunc') initialVals

                (_, finalSwarm) = runState (runPSOHelper maxIters')
SwarmState {...}
            -- output final message
        _ -> die ...
```

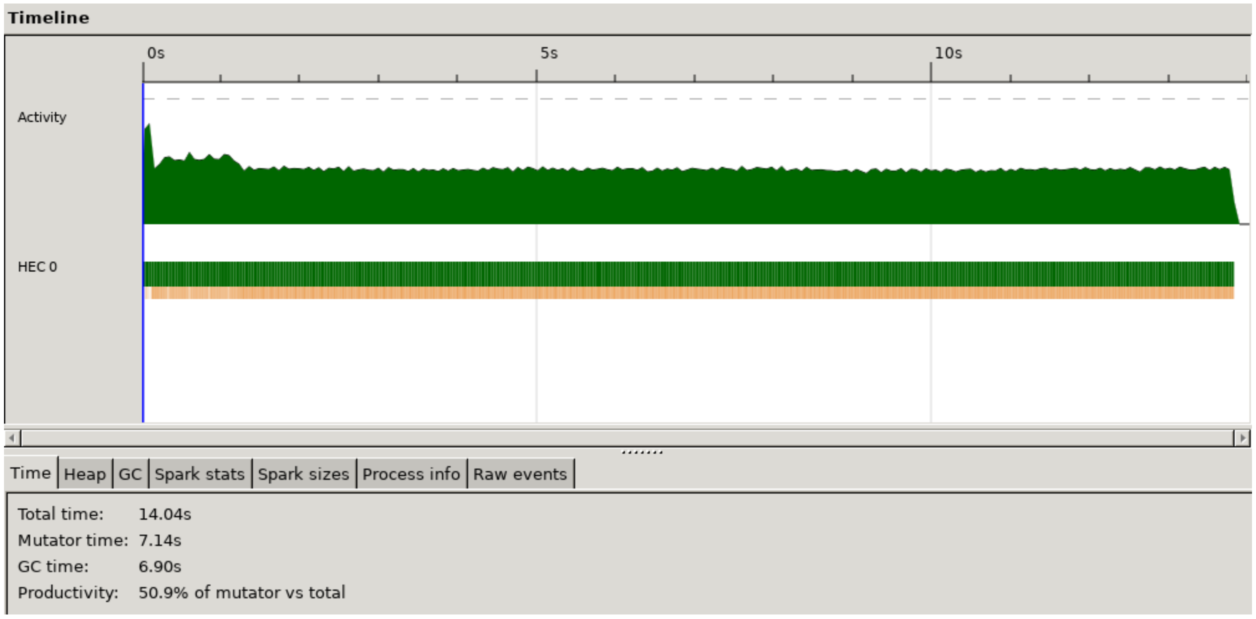
4 Experimental Results

For the testing of all of our methods, we initialize maxIters' = 1000, size' = 20000, posDims' = 2, veloDims' = 2, inertia' = 1, cognition' = 1.5, social' = 1.25. We will also always use Ackley's Function (objective function choice 4).

4.1 No Parallelization

```
 -- no Parallelization
 1 -> map (\a -> updateParticle a gB i c s o b) parts
```

When using this method, we use -N1, and so no parallelization occurs at all, and so it takes a decently long time:

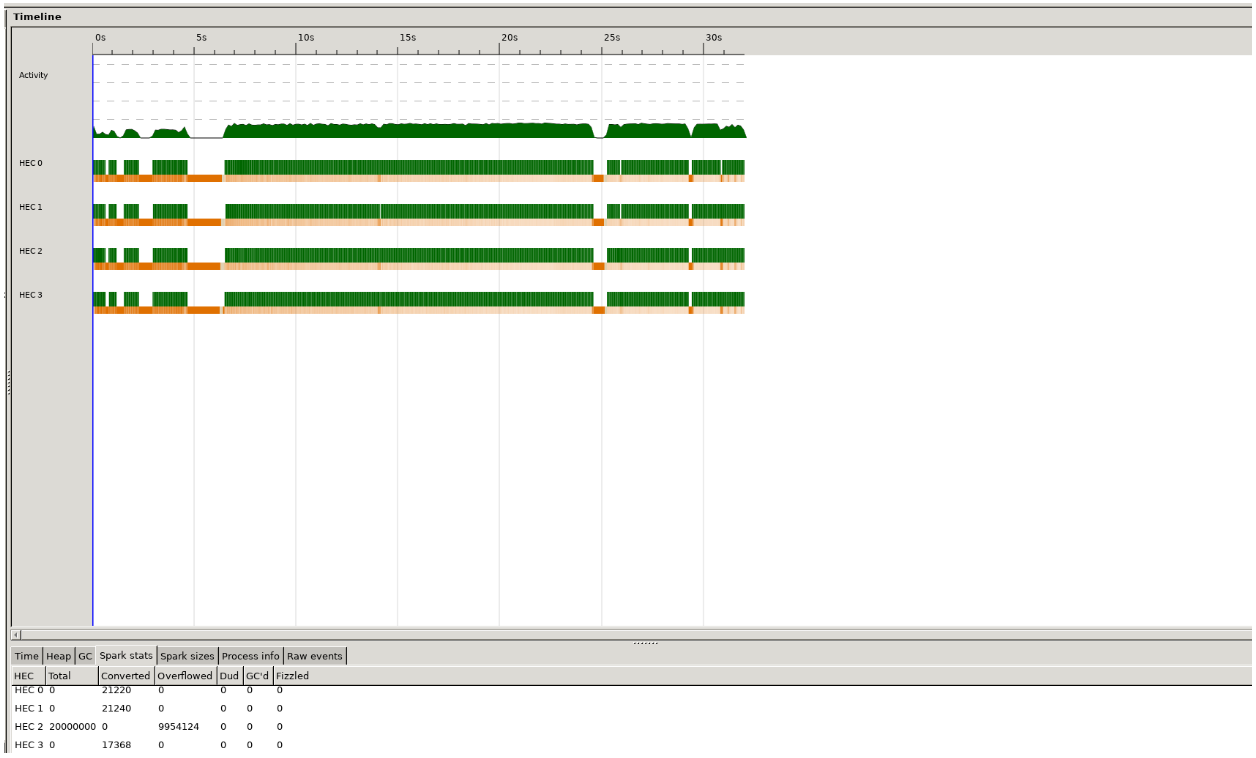


```
Total time:     14.04s
Mutator time:   7.14s
GC time:        6.90s
Productivity:   50.9% of mutator vs total
```

4.2 Using parList

```
  -- use of only parList
 3 -> map (\a -> updateParticle a gB i c s o b) parts
      `using` parList rdeepseq
```

For using parList with rdeepseq with -N4, we obtain an interesting result. Our implementation of parList simply pushes all the sparks onto the final core, and thus the runtime is significantly blown up. Due to this, there is an incredible amount of overflow on that final core, and the overhead is simply too high to even return a threadscope readable result. This is due to a simple and naive implementation of parList.



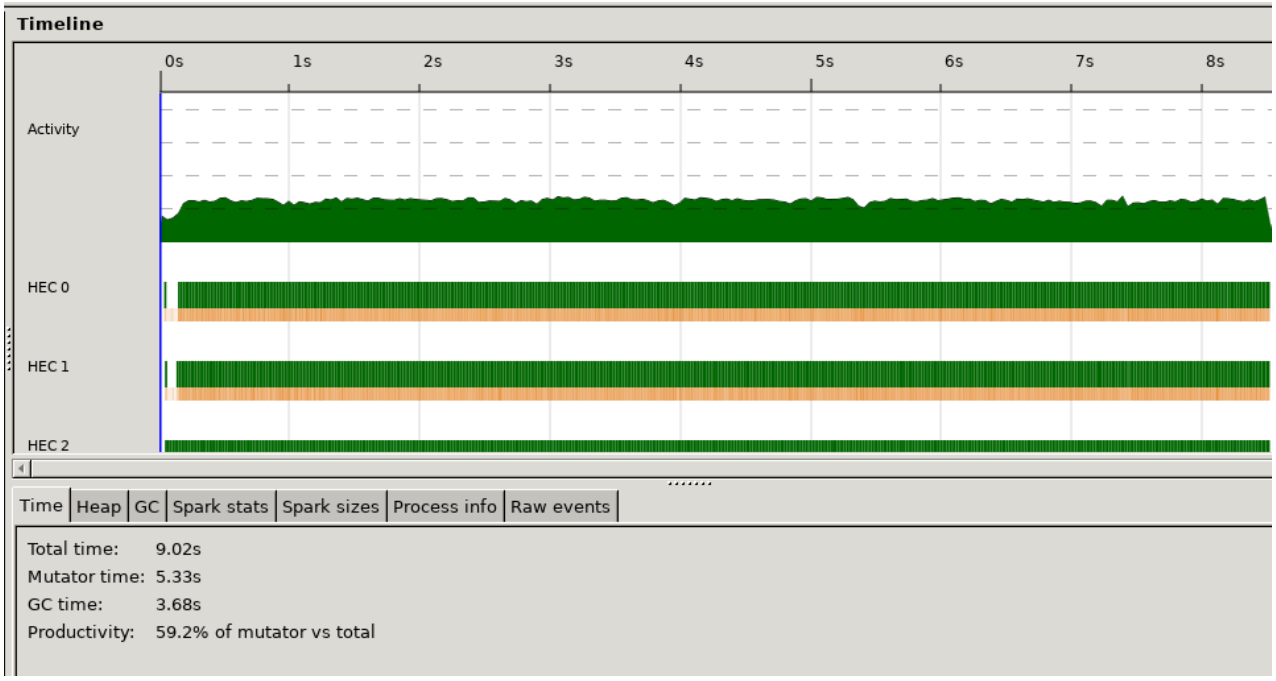
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
HEC 0	0	21220	0	0	0	0
HEC 1	0	21240	0	0	0	0
HEC 2	20000000	0	9954124	0	0	0
HEC 3	0	17368	0	0	0	0

Due to a poorly optimized implementation not utilizing rpar or anything of the sort, our implementation ends up with a runtime of over 30 seconds. All of the sparks are pushed onto one core, and the result is an incredibly blown up runtime.

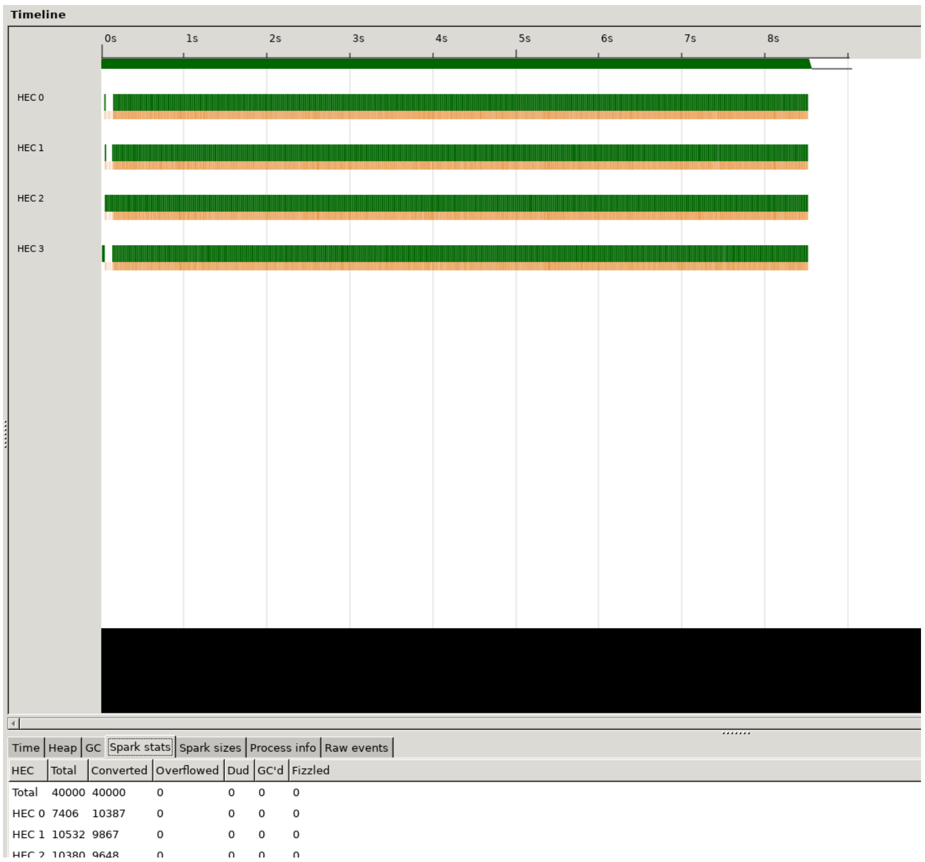
4.3 Using parListChunk

```
-- use of parListChunk
2 -> map (\a -> updateParticle a gB i c s o) parts
     `using` parListChunk (chunk swarm) rdeepseq
```

For using parListChunk with rdeepseq with -N4 and chunk size 500:



Total time: 9.02s
Mutator time: 5.33s
GC time: 3.68s
Productivity: 59.2% of mutator vs total



HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	40000	40000	0	0	0	0
HEC 0	7406	10387	0	0	0	0
HEC 1	10532	9867	0	0	0	0
HEC 2	10380	9648	0	0	0	0

4.4 Different Parameter Settings

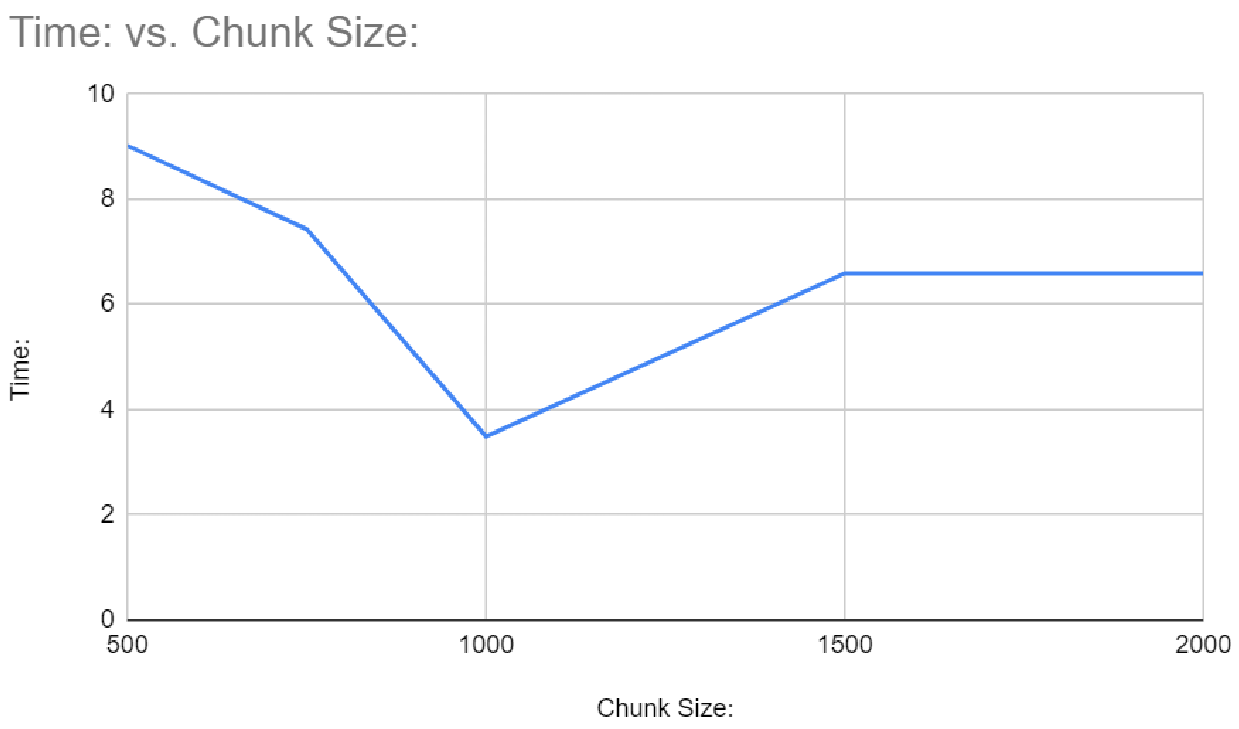
For -N4 results for various chunk sizes are as follows:

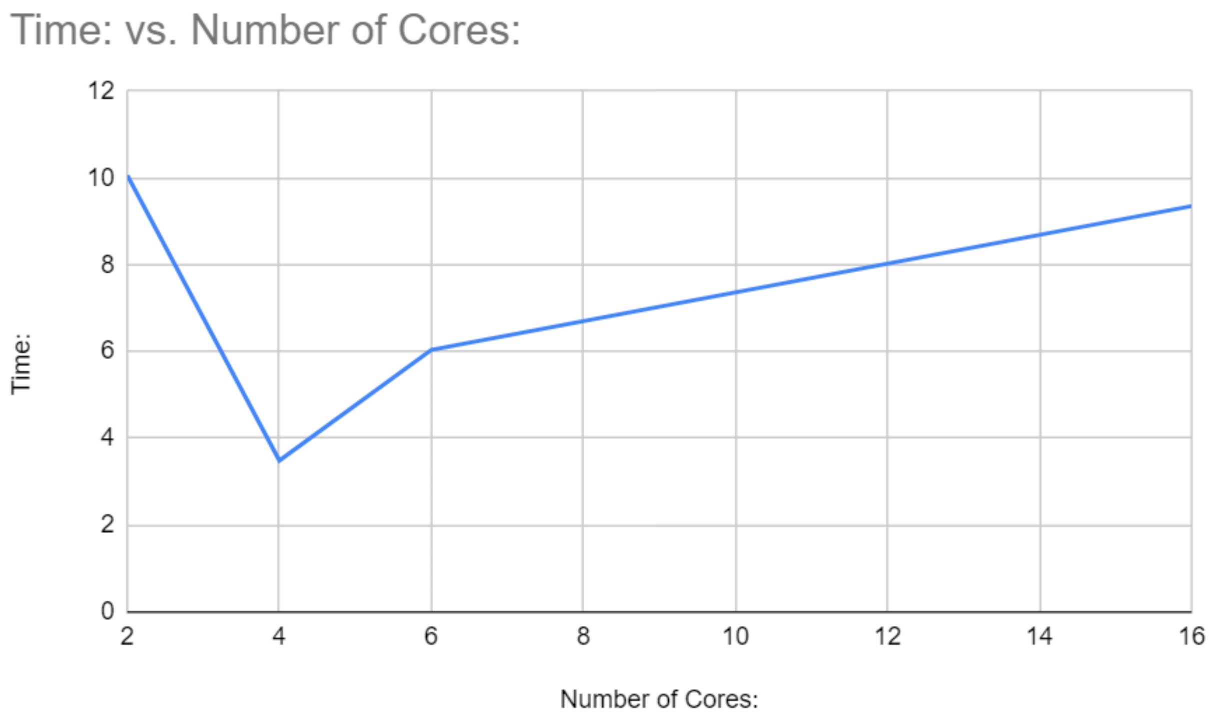
Chunk Size:	Time:
500	9.02
750	7.43
1000	3.49
1500	6.59
2000	6.59

For chunk size 1000, we utilize various number of cores:

Number of Cores:	Time:
2	10.06
4	3.49
6	6.04
16	9.35

The graphs are as follows:





For -N4 chunks it would seem the optimal number of chunks is about 100, given our initial conditions. The number of cores, for 1000 chunks, is definitely 4. This is just a product of us setting the optimal number of chunks for -N4 prior. Still, the type of data we obtain tells a lot.

5 Conclusion

Haskell's various parallel strategies and functional format lend to an incredible number of capabilities. If not properly implemented, however, as seen in our implementation of parList, the runtime is blown up. Although an incredibly powerful language, there are so many ways for parallelization to really blow up in your face. The extra stress due to the creation of various sparks can be detrimental if poorly implemented. Even under non-optimal conditions, however, our parListChunk implementation was able to easily outperform the non-parallelized implementation. Decently optimized, our parListChunk implementation was able to reach 3.49 seconds, while the non-parallelized clocked it 14.04 seconds. Depending on our choice of chunks and number of cores, our parallelization ends up at over 10 seconds. Choosing the correct initial conditions is so incredibly important in ensuring a decent runtime.

SwarmOps.hs:

```haskell
module SwarmOps (
    updatePos,
    updateVelo,
    updatePB,
    updateParticle,
    updateGB,
    particleInit,
    Particle(..),
    sphereFunction,
    quartFunction,
    rosenbrockFunction,
    ackleyFunction,
    rastriginObjective
) where

import qualified Data.List as L
import Control.Parallel.Strategies
import Control.DeepSeq

-- Sphere Function
sphereFunction :: [Double] -> Double
sphereFunction xs = sum $ map (^(2::Integer)) xs

-- Quartic function
quartFunction :: [Double] -> Double
quartFunction xs = sum $ map (^(5::Integer)) xs

-- Rosenbrock's Banana Function
rosenbrockFunction :: [Double] -> Double
rosenbrockFunction [x, y] = (1 - x)^(2::Integer) + 100 * (y -
x^(2::Integer))^(2::Integer)
rosenbrockFunction _ = error "Rosenbrock's function is defined for 2
dimensions only."

-- Ackley's Function
ackleyFunction :: [Double] -> Double
ackleyFunction xs = -20 * exp (a) - exp (b) + exp 1 + 20
```

```haskell
  where
    n = fromIntegral $ length xs
    a = -0.2 * sqrt (sum (map (\x -> x^(2::Integer)) xs) / n)
    b = sum (map (\x -> cos (2 * pi * x)) xs) / n

-- Rastrigin's Objective Function
rastriginObjective :: [Double] -> Double
rastriginObjective xs = 10 * fromIntegral (length xs) + sum
[x^(2::Integer) - 10 * cos (2 * pi * x) | x <- xs]

data Particle =  Particle {position :: [Double],
                           velocity :: [Double],
                           bestPos :: [Double],
                           bestFitness :: Double}

instance NFData Particle where
    rnf (Particle pos velo bPos bFit) =
        rnf pos `deepseq`
        rnf velo `deepseq`
        rnf bPos `deepseq`
        rnf bFit

-- update particle pos, and outputs new pos vector
updatePos :: Particle -> Particle
updatePos particle =
    particle {position = zipWith f' pos velo}
        where pos  = position particle
              velo = velocity particle
              f' p v  = p + 0.5*v

-- update particle velo, and outputs new velo vector
updateVelo :: Particle -> [Double] -> Double -> Double -> Double ->
Particle
updateVelo particle globalBest' inertia' cognitive' social' =
    particle {velocity = L.zipWith4 (\v p pB gB -> inertia' * v +
cognitive' * (pB - p) + social' * (gB - p) )
             (velocity particle) (position particle) (bestPos
particle) globalBest'}
```

```
--updates the personal best of that particle
updatePB :: Particle -> ([Double] -> Double) -> Particle
updatePB particle objFunc'
    |fit < bestFitness particle = particle {bestFitness = fit,
                                            bestPos     = position
particle}
    |otherwise                  = particle
        where fit = objFunc' $ position particle

-- combines both updateVelo and pos to update the particle
updateParticle :: Particle -> [Double] -> Double -> Double -> Double
-> ([Double] -> Double) -> Particle
updateParticle particle globalBest' inertia' cognitive' social'
objFunc' =
    updatePB (updatePos (updateVelo particle globalBest' inertia'
cognitive' social')) objFunc'

-- updates the global best of the swarm for the objective function
updateGB :: [Particle] -> [Double]
updateGB particles' = position (L.minimumBy (\a b -> compare
(bestFitness a) (bestFitness b)) particles')
                        `using` parList rseq



-- initialize particle with random value within given ranges
particleInit :: [Double] -> [Double] -> ([Double] -> Double) ->
Particle
particleInit initPosition initVelocity objFunc' =
    Particle {position = initPosition, velocity = initVelocity,
              bestPos = initPosition, bestFitness = objFunc'
initPosition}
```

Main.hs:

```haskell
module Main (
    main
) where

import SwarmOps as S
import System.Random
import Control.Monad
import Control.Parallel.Strategies
import System.Environment
import System.Exit
import Control.Monad.State

-- define SwarmState
data SwarmState = SwarmState
                   {particles :: [Particle],
                    size :: Int,
                    maxIters :: Int,
                    globalBest :: [Double],
                    inertia :: Double,
                    cognition :: Double,
                    social :: Double,
                    posDims :: Int,
                    veloDims :: Int,
                    chunk :: Int,
                    choice :: Int,
                    objFunc :: ([Double] -> Double)}

-- iterate through one instance of the swarm
iterateSwarm :: State SwarmState ()
iterateSwarm = do
    swarm <- get
    let gB = globalBest swarm
        i  = inertia swarm
        c  = cognition swarm
        s  = social swarm
        o  = objFunc swarm
        m  = fromIntegral(maxIters swarm)
        i' = i * ((m-1)/m)
        parts  = particles swarm
        upParts =
            case (choice swarm) of
                -- no Parallelization
                1 ->
                    map (\a -> updateParticle a gB i c s o) parts
                -- use of parListChunk
                2 ->
                    map (\a -> updateParticle a gB i c s o) parts
                    `using` parListChunk (chunk swarm) rdeepseq
                -- use of only parList
                3 ->
                    map (\a -> updateParticle a gB i c s o) parts
                    `using` parList rdeepseq
                _ -> []
        upGB = updateGB $ upParts
    modify $ \up -> up { particles = upParts,
                         globalBest = upGB,
                         inertia = i'
                         }
    return ()

-- Helper function for recursive PSO iterations
runPSOHelper :: Int -> State SwarmState ()
runPSOHelper 0 = return ()
runPSOHelper n = do

    -- Update the swarm
    iterateSwarm

    -- Continue the iterations
    runPSOHelper (n - 1)

main :: IO ()
main = do
    args <- getArgs
    progName <- getProgName
    putStrLn "<Parallelization Choices>"
    putStrLn "<1:No Parallelization>"
    putStrLn "<2:Use of parListChunk>"
    putStrLn "<3:Use of purely parList>\n"
    putStrLn "<Optimization Examples>"
    putStrLn "<1:Sphere>"
    putStrLn "<2:Quartic>"
    putStrLn "<3:Rosenbrock>"
    putStrLn "<4:Ackley>"
    putStrLn "<5:Rastrigin>\n"
    case args of
        [parChoice, optEx] -> do
            let size' = 20000
                posDims' = 2
                veloDims' = 2
                maxIters' = 1000
                inertia' = 1
                cognition' = 1.5
                social' = 1.25
                chunk' = 1500
                optEx' = read optEx::Int
                choice' = read parChoice::Int

            initialPos <- replicateM size' $ replicateM posDims'
(randomIO)
            initialVelo <- replicateM size' $ replicateM veloDims'
(randomRIO (-1, 1))
            let initialVals = zip initialPos initialVelo
                objFunc'
                    |optEx' == 1  = sphereFunction
                    |optEx' == 2  = quartFunction
                    |optEx' == 3  = rosenbrockFunction
                    |optEx' == 4  = ackleyFunction
                    |optEx' == 5  = rastriginObjective
                    |otherwise    = error "Usage: swarm-test-exe
<Parallelization Choice (1,2,3)> <Optimization Example (1,2,3,4,5)>"
                particles' = map (\(initP, initV) -> particleInit
initP initV objFunc') initialVals
```

```
                (_, finalSwarm) = runState (runPSOHelper maxIters')
SwarmState
                                  { particles = particles',
                                    size = size',
                                    maxIters = maxIters',
                                    globalBest = updateGB particles',
                                    inertia = inertia',
                                    cognition = cognition',
                                    social = social',
                                    posDims = posDims',
                                    veloDims = veloDims',
                                    chunk = chunk',
                                    choice = choice',
                                    objFunc = objFunc'}

            putStrLn "Final Global Best Position:"
            print $ globalBest finalSwarm
            putStrLn "Final Global Best Fitness:"
            print $ (objFunc finalSwarm) $ globalBest finalSwarm

        _ -> die $ "Usage: " ++ progName ++ " <Parallelization Choice
(1,2,3)> <Optimization Example (1,2,3,4,5)>"

-- test using stack run -- swarm-test-exe +RTS -N4 -l -RTS n m
-- threadscope swarm-test-exe.eventlog
-- must do so in ubuntu after going to cd "/mnt/c/Users/vindi/Haskell
Stuff/project/swarm-test"
```