GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

PyGMA : Pythonic Genetic MPI parallelized Algorithm

Course Practical Course on High-Performance Computing
Presenter Winfried Oed
Supervisor Aasish Kumar Sharma

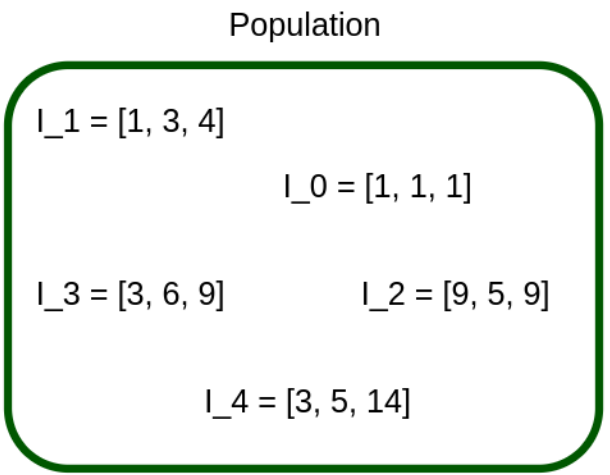
10 July 2023

Agenda

- Genetic Algorithm principles
- PyGMA implementation
- Usage example
- MPI Implemetation
- Conclusion

Genetic Algorithm

- GA combines
  - Darwin’s theory of evolution
  - Mendels theory of genetics
- Population
  - Individuals
    - Represent solution
    - Have fitness
  - Evaluation
  - Mutation
  - Recombination



Y = w_0 * 3 + w_1 * 1 + w_2 * 2

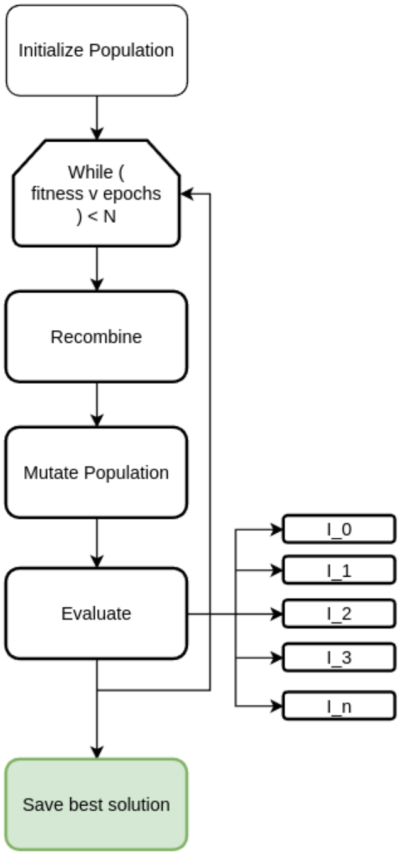
How to choose w_n such that Y = 42?

I_3 = 3×3+6×1+9×2 = 33

error = | 33 - 42 | = 9

fitness = 1.0 / 9 = 0.111

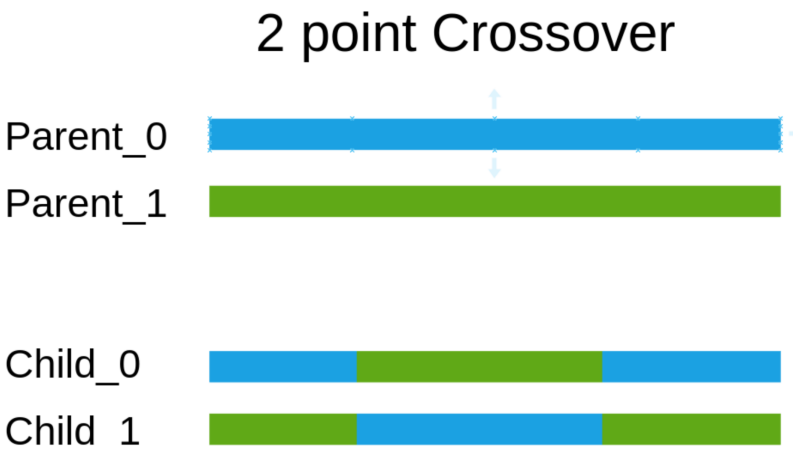
Algorithmic flow



Genetic Operators

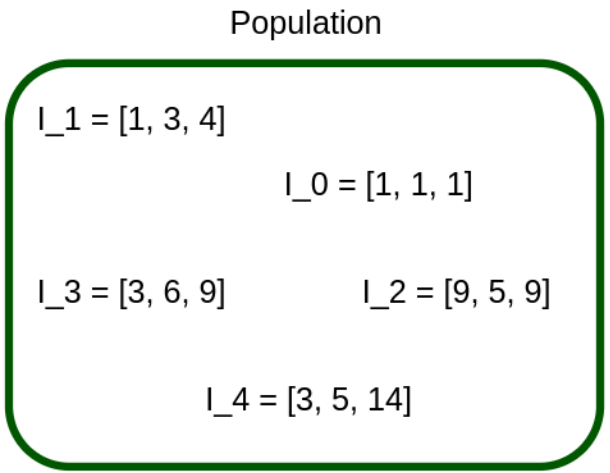
- Mutation
  - Random change
  - 1001 → 1000
- Crossover
  - 1, 2, k point
- Bitwise operators
  - AND, OR, XOR

- Chosen Operators depend on the problem



Experiments

- The experiment is the task which the GA should solve



Y = w_0 * 3 + w_1 * 1 + w_2 * 2

How to choose w_n such that Y = 42?

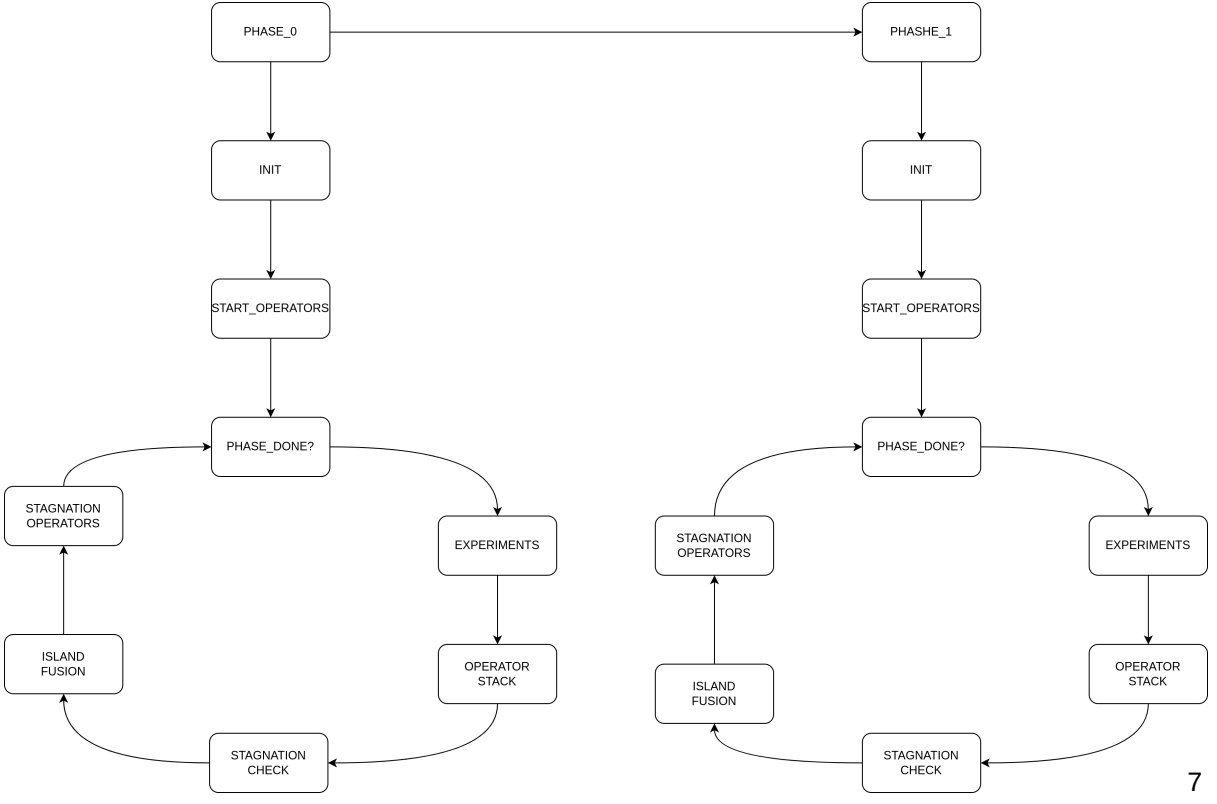
I_3 = 3×3+6×1+9×2 = 33

error = | 33 - 42 | = 9

fitness = 1.0 / 9 = 0.111

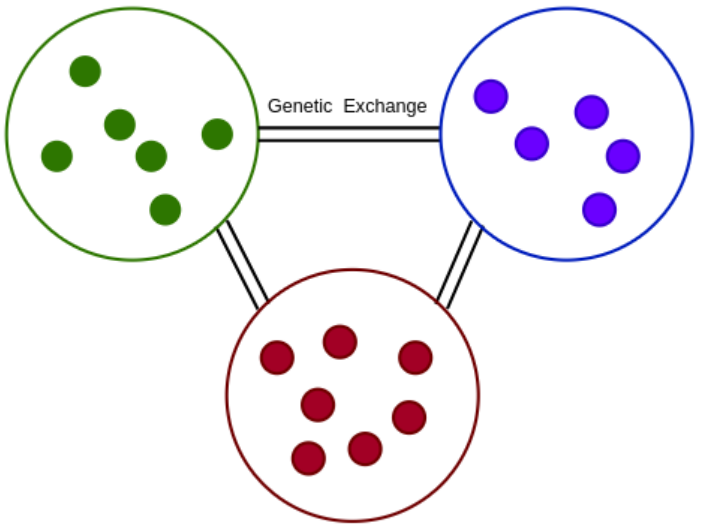
Evolutionary Phase

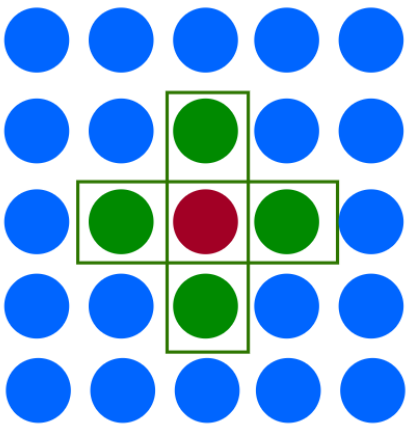
- Idea:
  - do not solve complex problems directly
  - Solve a simple version first
  - Then a more complex one
  - …



Evolution Models

- Island’s
  - Island fusion
- Cellular
- Hybrid
  - <Island, Cellular>



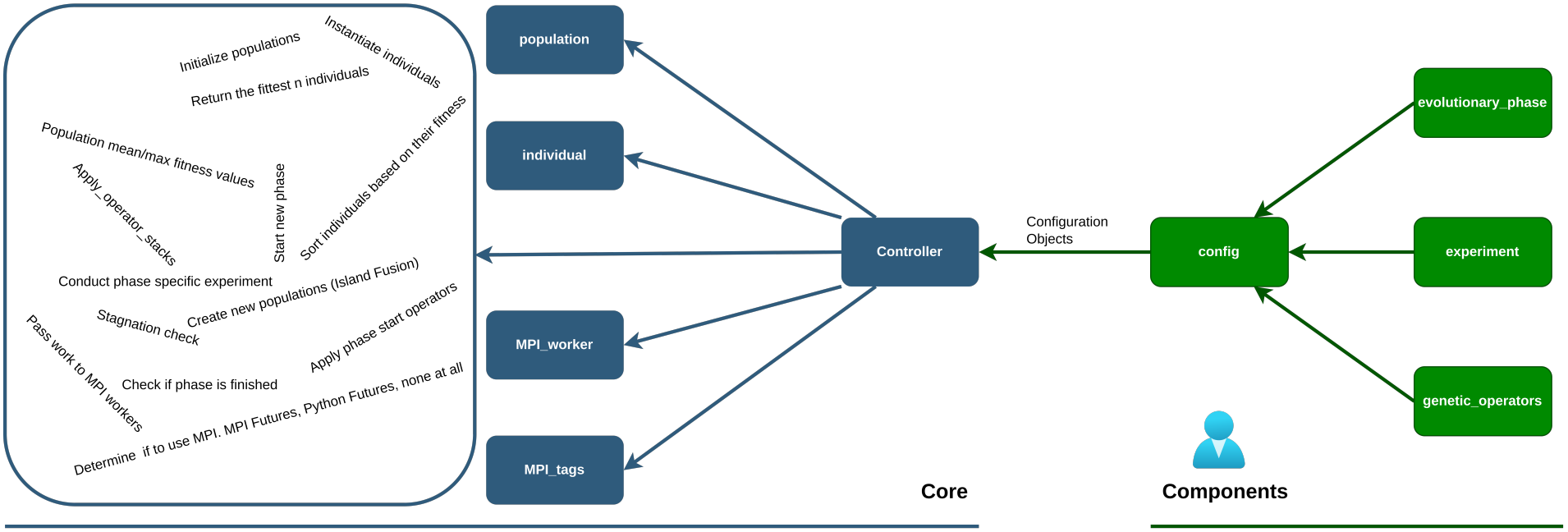


PyGMA

PyGMA

- Pythonic Genetic MPI parallelize’d Algorithm
  - Handles the important GA parts
- User interaction
  - Class interfaces
    - Genetic Operators
    - Experiments
    - Evolutionary Phases
  - Config

PyGMA


Initialize populations
Instantiate individuals
Return the fittest n individuals
Population mean/max fitness values
Apply_operator_stacks
Start new phase
Sort individuals based on their fitness
Conduct phase specific experiment
Create new populations (Island Fusion)
Stagnation check
Apply phase start operators
Pass work to MPI workers
Check if phase is finished
Determine if to use MPI. MPI Futures, Python Futures, none at all
population
individual
MPI_worker
MPI_tags
Controller
Configuration Objects
config
evolutionary_phase
experiment
genetic_operators
Core
Components

Usage Example

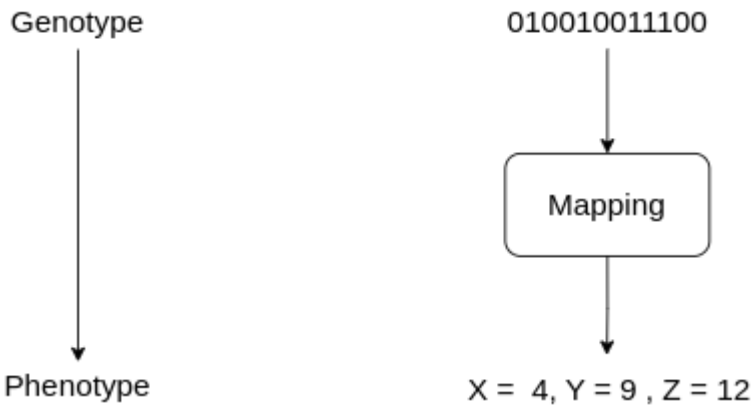
- Function optimisation
  - F = x*2 + y*3 + z
  - How to choose x,y,z such that F = 424242 ?
- Steps
  - Define experiment
  - Define evolutionary phase (optional)
  - Define genetic operators (optional)
  - Create configuration
  - Run PyGMA

Define Experiment

```python
class Function_optimisation_experiment(Experiment):
    """
    Find values for the function such that it matches a certain output
    """

    def f(self, x, y, z):
        return x*2 + y*3 + z

    def conduct(self, gene) -> float:
        # split gene into blocks that are binary representation
        # of the numbers
        numbers = np.split(gene, 3)
        # convert the numbers to ints
        x = bool2int(numbers[0])
        y = bool2int(numbers[1])
        z = bool2int(numbers[2])

        # define the objective output for our function
        output = 424242

        # calculate the output
        r = self.f(x, y, z)

        # calc error and fitness
        error = abs(r-output)

        # if error is small fitness is high
        # make sure that if error can not be 0 :)
        fitness = 1.0/(error + 0.00000000000000001)

        # print(f'error: {error} fitness {fitness}')
        return fitness
```

Define Phase

```python
class Simple_evolutionary_phase(Evolutionary_phase):
    def completed(self, population_fitness: list, epochs: int) -> bool:
        """
         Called to check if the phase has reached its end.
        """
        max_fitness = max(population_fitness)
        if max_fitness > 0.9:
            return True

        if epochs > 5000:
            return True

        return False
```



Define Genetic Operators

```
class Mutation_operator(Genetic_operator):
    """
    This class represents a mutation operator.
    """

    def __init__(self, mutation_rate, rng_seed=9):
        self.mutation_rate = mutation_rate
        self.rng = np.random.default_rng(rng_seed)

    def operate(self, old_population, new_population):
        """
        Apply mutation to all genes of the current op stack.
        """
        for gene in new_population:
            # mutation indexes
            # having at least one mutation
            mutations = int(self.mutation_rate*gene.size)
            if mutations == 0:
                mutations = 1
            indexes = self.rng.integers(low=0, high=gene.size, size=mutations)
            # flip the bits at indexes
            # note, ~ is like np.invert()
            gene[indexes] = ~gene[indexes]
        return new_population
```

Make Configuration

- Instantiation of
  - Genetic Operators
  - Experiment
  - Evolutionary Phases
- Genetic Operator stack/Num Populations
- Individuals per population
  - Genome length
- Initialisation of Genes (predef, random)
- Use MPI ?

```
# ----------------------------
# Genetic operator definition
# ----------------------------
# Genetic operators can be used for:
# Populations
# Genetic Phases

OP_MUTATION = Mutation_operator(0.06)
OP_REMOVAL = Removal_operator(6)
OP_CROSSOVER = Single_point_crossover_operator(6)
```

```
# -------------
# Populations
# --------------
# how many island populations to use
# is defined by the genetic operators stack length
# used to recombine/produce, mutate, extend...
'genetic_operator_stack': [
    # pop 0
    [OP_REMOVAL, OP_CROSSOVER],
    # pop 1
    [OP_REMOVAL, OP_CROSSOVER, OP_MUTATION],
    # pop 2
    [OP_REMOVAL, OP_REMOVAL, OP_REMOVAL, OP_MUTATION, OP_CROSSOVER, OP_CR
],
```

Run Pygma

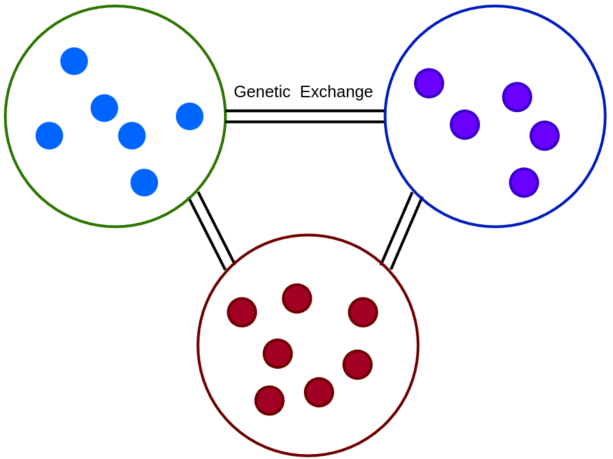
- mpiexec -n 3 python PyGMA.py

- Evolutionary phase Phase_0
- epoch 0, Pop max fitness [9.925775054145103e-07, 2.12027020723521e-06, 1.6586828731041255e-06]
- epoch 1, Pop max fitness [1.3549254249046132e-06, 1.545657167057718e-06, 1.9955061202172707e-06]
- epoch 2, Pop max fitness [1.3549254249046132e-06, 1.585273443816324e-06, 0.0005668934240362812]
- epoch 3, Pop max fitness [2.2883504654504845e-06, 3.542180282807674e-06, 3.122268015486449e-05]
- …
- epoch 209, Pop max fitness [3.054386404315237e-06, 0.0002605523710265763, 0.0004068348250610252]
- epoch 210, Pop max fitness [3.054386404315237e-06, 0.00011841326228537596, 4.1221814584278e-05]
- epoch 211, Pop max fitness [3.054386404315237e-06, 0.00013356484573260317, 0.0004730368968779565]
- epoch 212, Pop max fitness [3.054386404315237e-06, 9.994003597841295e-05, 1e+17]
- Finished phase Phase_0
- ---
- stopping mpi workers
- Evolve finished

Parallelization

Parallelization

- Idea
  - If there are multiple populations (Islands), compute each on its own node
  - Genetic operators are applied in parallel
  - But
    - Normally one does not want to use to many Islands
    - Need to evaluate individual genes anyway and as such distribute them
- Idea
  - Applying Genetic operators is fast
  - Fitness evaluation takes time
  - => Distribute individual genes to workers that will evaluate the fitness



Parallelization schemata

- None
- Local Processes
  - Python’s ProcessPoolExecutor
- Message Passing Interface
  - MPI calls (utilizing mpi4py)
  - mpi4py MPIPoolExecutor

MPI implementation

```python
# will use the cached module and as such the global intercomm
from mpi4py import MPI
from core.mpi_tags import mpi_tags
comm = MPI.COMM_WORLD
world_size = comm.Get_size()

# while we have data distribute it to free workers
# collect finished data as well
# a gene is computed once the result is written back
genes_to_compute = len(worker_tuples)
while genes_to_compute > 0:
    # check all workers
    for worker_index in range(1, world_size):

        # check if worker can compute new data (Is idle) and we have data to compute
        if worker_tuples and comm.iprobe(source=worker_index, tag=mpi_tags.IDLE):
            # eat up that message to clean the pipeline
            comm.recv(source=worker_index, tag=mpi_tags.IDLE)
            # pop the next data to distribute
            data = worker_tuples.pop()
            # signalling the worker that it should continue to work
            comm.send(True, dest=worker_index,
                      tag=mpi_tags.CONTINUE_PROCESSING)
            # send data to worker
            comm.send(data, dest=worker_index,
                      tag=mpi_tags.DATA)

        # check if we can retrieve a result from the worker
        if comm.iprobe(source=worker_index, tag=mpi_tags.DATA_RETURN):
            # retrieve the result
            w_result = comm.recv(
                source=worker_index, tag=mpi_tags.DATA_RETURN)
            # write the fitness result into the appropriate individual
            self.populations[w_result[0]
                             ].individuals[w_result[1]].fitness = w_result[2]
            # note that we have computed a gene
            genes_to_compute -= 1
```

```python
# worker
# work until stop is signalled
while True:
    # print(f'worker {rank} loop')
    # signaling that we are waiting for new work
    self.comm.isend(None, dest=0, tag=mpi_tags.IDLE)

    # we receive a bool if we shall to continue to process
    # to not eat all CPU ressources we sleep a little if no transmission
    # but this will slow down tremendously,
    # don't sleep when you are supposed to be present
    if self.sleep_time > 0:
        while not self.comm.iprobe(source=0, tag=mpi_tags.CONTINUE_PROCESSING):
            # sleep
            time.sleep(self.sleep_time)
    continue_processing = self.comm.recv(
        source=0, tag=mpi_tags.CONTINUE_PROCESSING)

    # if we shall not continue stop the worker
    if not continue_processing:
        break

    # get the working data
    # format will be (pop_index, indiv_index, phase_index, gene)
    data = self.comm.recv(source=0, tag=mpi_tags.DATA)

    # extract data
    pop_index = data[0]
    indiv_index = data[1]
    phase_index = data[2]
    gene = data[3]

    # instantiate the experiment and evaluate the individual
    experiment = self.config['evolutionary_phases'][phase_index].experiment

    # conduct the experiment
    fitness = experiment.conduct(gene)

    # return the result tuple
    self.comm.send((pop_index, indiv_index, fitness),
                   dest=0, tag=mpi_tags.DATA_RETURN)
```

Future Work

- Stagnation check
- Hardware evolving experiment
  - Experiment
  - Operators
  - Phases
- Runtime analysis (Single process, multi process, MPI)

Conclusion

- GA Algorithm works (MPI)
- Defining Fitness function for hardware evolving is not simple
- Diagram's help a lot during implementation