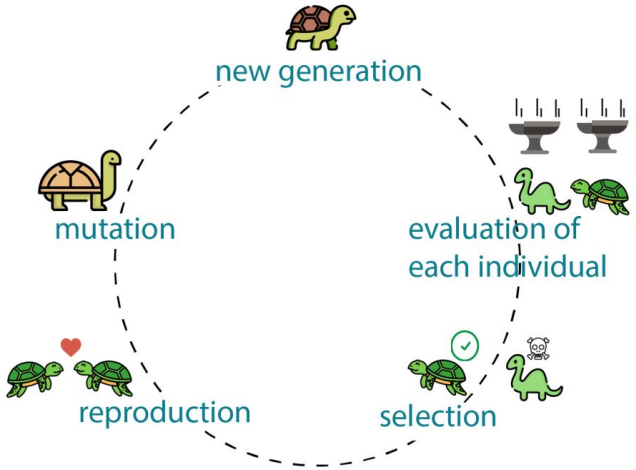
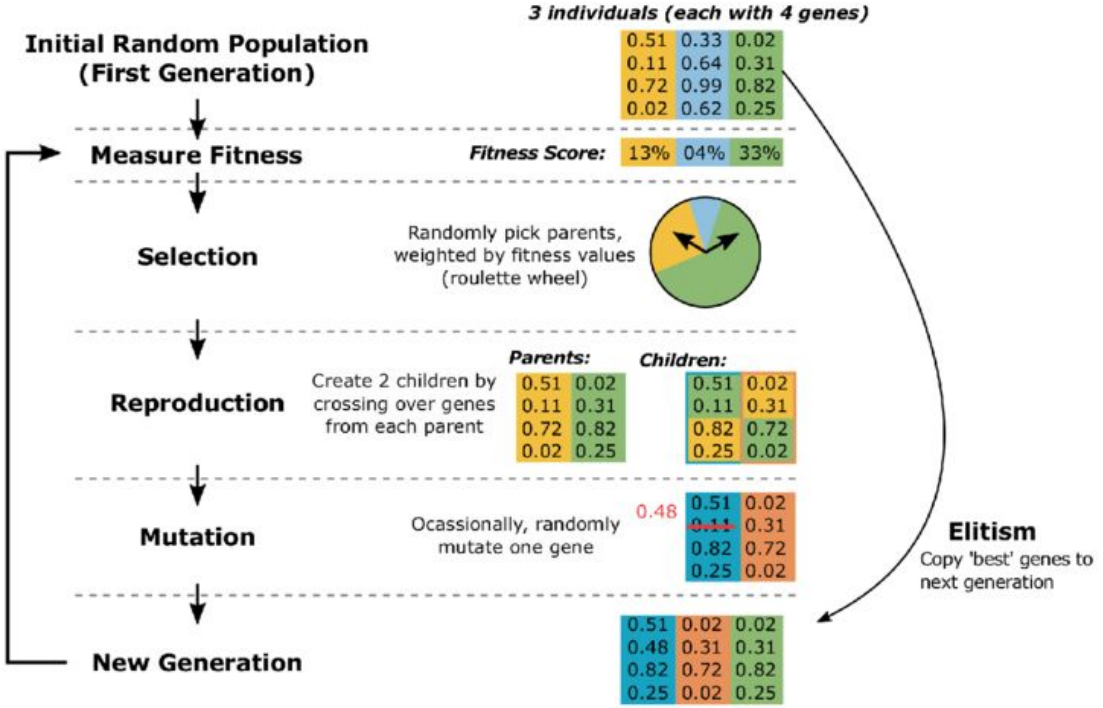
Parallel Genetic Algorithm

Team 41　葛奕宣 鄔培勇 蔡芝泰

Introduction

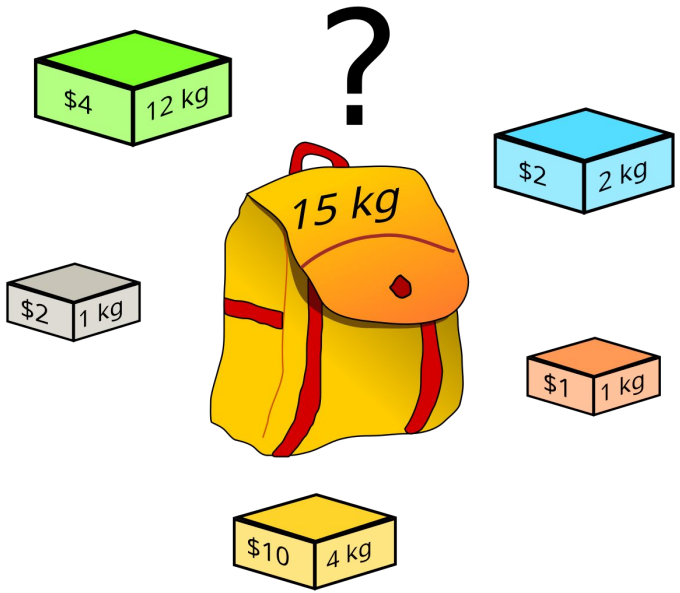
- Genetic Algorithm





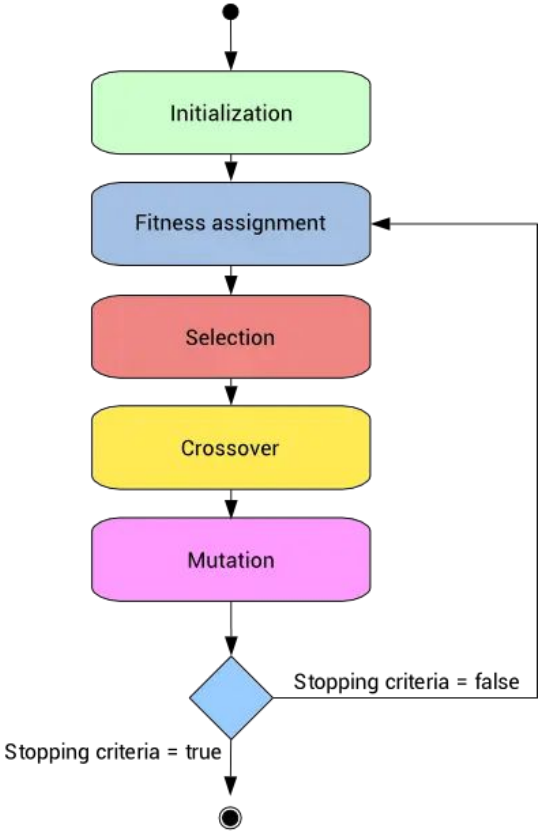
Introduction

- Approximate Algorithm
  - 0/1 knapsack problem
    - use 10010 to represent
  - traveling salesman problem
  - Neural Network Optimization

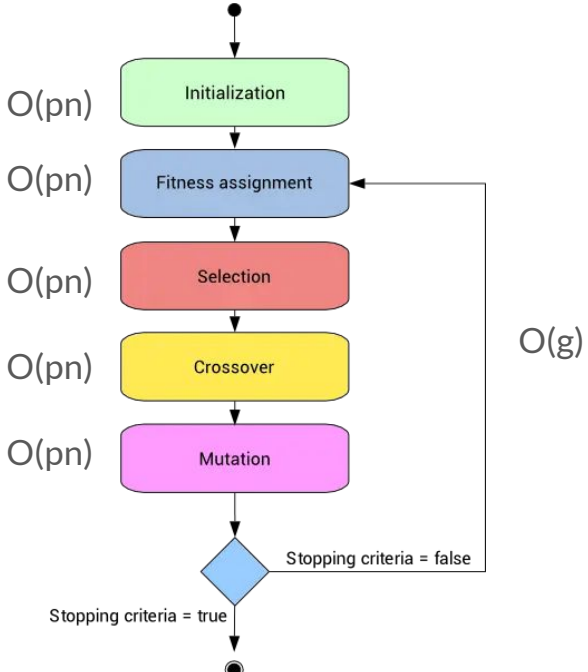


Introduction

- Population (**p**)
  - Selections
- Population.Gene (**n**)
  - Indicate which item taken (01101)
- Fitness function
  - How good is this selection?
- Generation (**g**)
  - Iteration time



Time Complexity Analysis



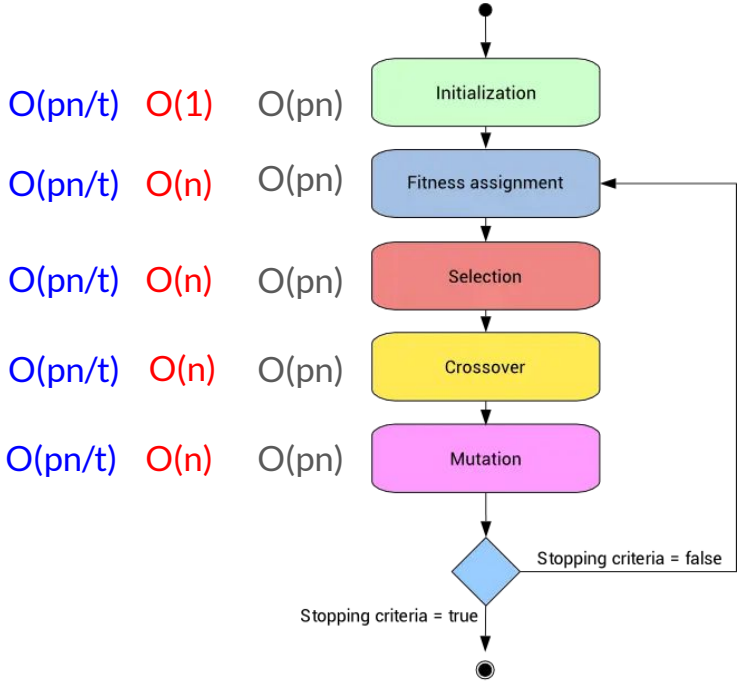
Total: **O(gpn)**

Parallel Strategy

```
initialize_population(population);
evaluate_population(population);

while (generation < GENERATIONS) {
    selection(population, new_population);
    for (int i = 0; i < POP_SIZE; i += 2) {
        crossover(new_population[i], new_population[i + 1], &population[i], &population[i + 1]);
        mutate(&population[i]);
        mutate(&population[i + 1]);
    }
    evaluate_population(population);
    generation++;
}
```

Time Complexity Analysis

- CUDA
- pthread, omp, mpi+omp
  - Total # of threads (t)



Total: **O(gn)** **O(gpn/t)**

Challenge – Random function

- In C, everytime rand() is called, it will fetch the random seed setup at the beginning(srand()).
- This would cause significant concurrency overhead.
  - Each thread read/write the random state.

Challenge – Random function

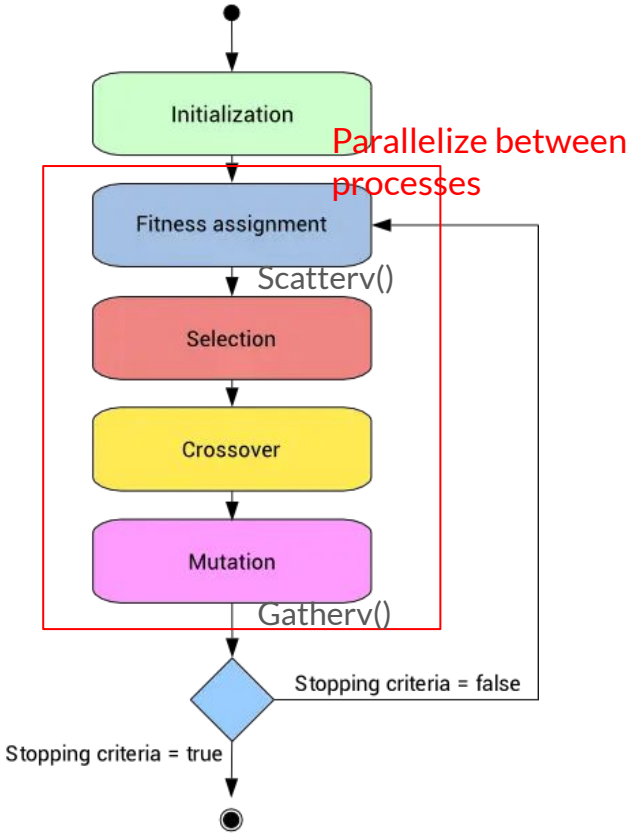
- Cuda Solution:
  - Create a curandState variable for each thread.
  - Initialize with the same seed at the beginning

```
__global__ void setup_random_states(curandState *states, unsigned long seed) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    curand_init(seed, tid, 0, &states[tid]);
}
```

- Pthread Solution:
  - Create local random seed
  - Use rand_r(&local_seed) instead of rand()

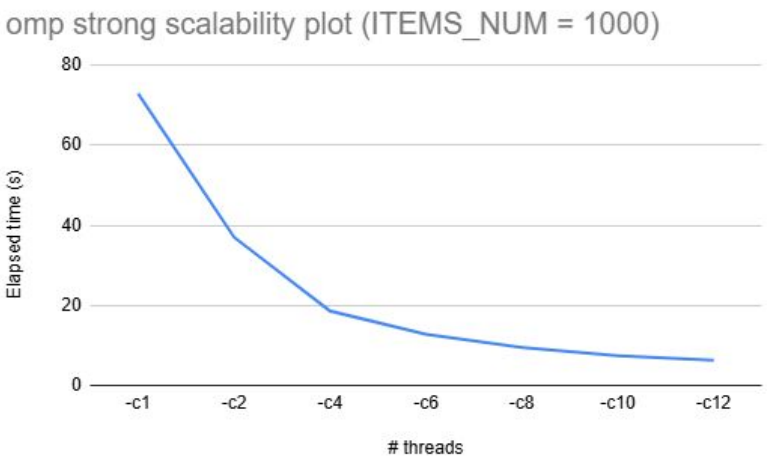
Parallel Strategy (MPI)

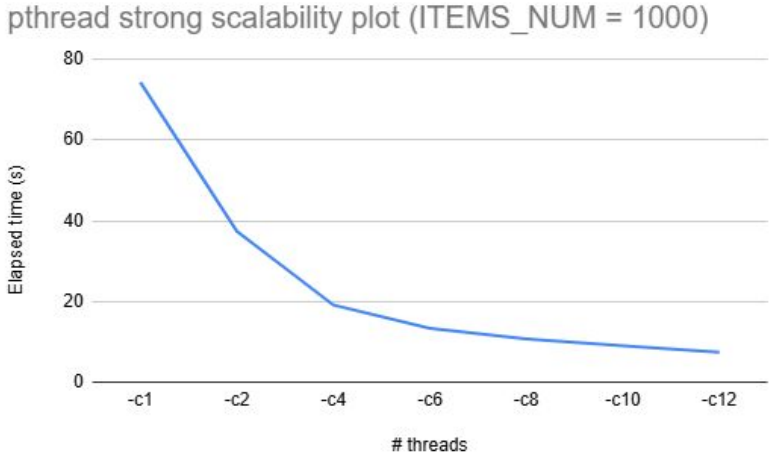
```
for (int generation = 0; generation < GENERATIONS; generation++) {
    selection(population, new_population);
    for (int i = 0; i < POP_SIZE; i++) {
        memcpy(global_genes + i * ITEMS_NUM, new_population[i].genes, ITEMS_NUM * sizeof(int));
    }

    MPI_Scatterv(global_genes, recv_counts, displacements, MPI_INT,
                 local_genes, local_size * ITEMS_NUM, MPI_INT, 0, MPI_COMM_WORLD);

    for (int i = 0; i < local_size; i += 2) {
        crossover_array(local_genes + i * ITEMS_NUM, local_genes + (i + 1) * ITEMS_NUM, local_new_genes
        mutate_array(local_new_genes + i * ITEMS_NUM);
        mutate_array(local_new_genes + (i + 1) * ITEMS_NUM);
    }

    MPI_Gatherv(local_new_genes, local_size * ITEMS_NUM, MPI_INT,
                global_genes, recv_counts, displacements, MPI_INT, 0, MPI_COMM_WORLD);

    for (int i = 0; i < POP_SIZE; i++) {
        population[i].genes = global_genes + i * ITEMS_NUM;
    }
    evaluate_population(population);
}
```



Evaluation

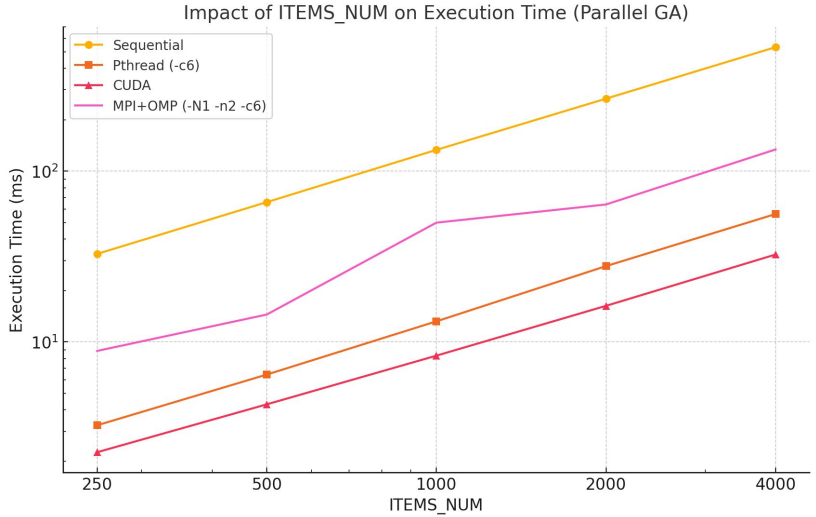
- Strong Scalability





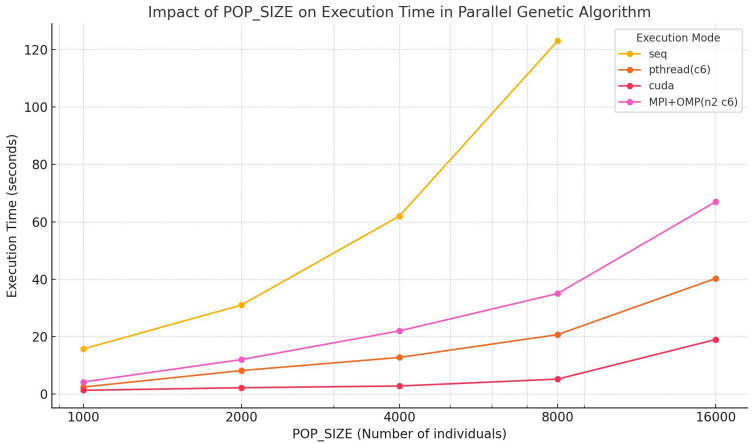
Evaluation - Impact of ITEMS_NUM

- **All implementation shows linear relationship between ITEMS_NUM and execution time**
- **Our algorithm doesn’t parallel this parameter**



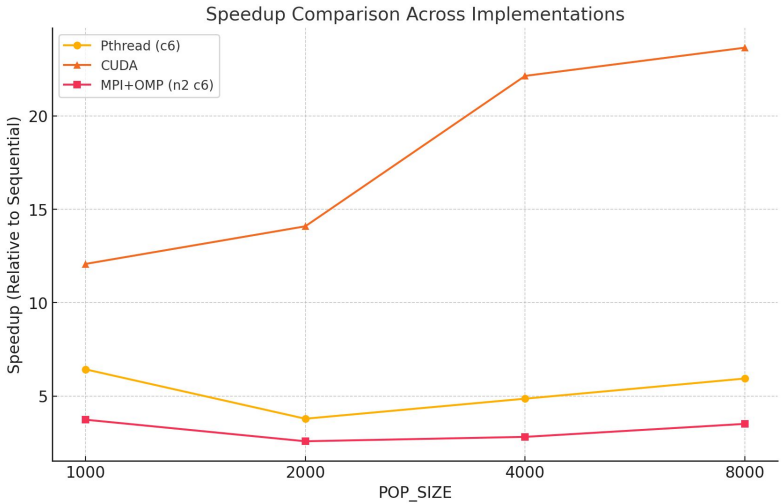
Evaluation - Impact of POP_SIZE

- **Sequential**: Linear relationship with `POP_SIZE`, poor scalability.
- **CUDA**: No clear linear trend; efficient for small-medium sizes but slows at large scales.
- **Pthread/MPI+OMP**: Scales as `POP_SIZE / #threads`, effective but limited by overhead at large sizes.



Evaluation - Overall Speedup compare to Seq

- **CUDA**: Highest speedup for up to 23x, excellent scalability.
- **Pthread (c6)/ MPI+OMP**: Stable scalability.



Conclusion

- **Summary of Work**
  - Implemented Parallel Genetic Algorithms using **CUDA, OMP**, **Pthread**, **MPI**, **MPI+OMP**.
  - Addressed challenges like random function overhead efficiently.
- **Key Findings**
  - **CUDA** achieved the highest speedup (~23x), excelling for small-medium workloads.
  - **CPU** solutions provided stable scalability but faced overhead at larger scales.
  - Linear execution time observed with **ITEMS_NUM** and **POP_SIZE**.