并行化遗传算法求解 TSP 问题的研究与实现

周宝航
2120190442
zhoubaohang@dbis.nankai.edu.cn

摘要 旅行商推销员问题（Travelling salesman problem）是组合优化中的一个 NP 难问题，在运筹学和理论计算机科学中非常重要。随着问题规模的增大，精确算法求解该类问题变得无能为力。因此，国内外学者重点研究近似算法或启发式算法以求解该问题。而遗传算法作为一种启发式算法可以快速地将解空间中的全体解搜索出来，并且全局搜索能力强，克服了其他算法的加速下降陷阱问题。当问题的规模和复杂程度不断增加以后，串行化遗传算法收敛到全局最优所花费的时间也更长。借助遗传算法天然的并行性，我们采用不同的并行策略以加速算法求解全局最优值。遗传算法的难点在于采用何种方法既保证优良个体的同时，保证群体的多样性。因此，最常采用的并行策略为初始化多种群同时进行算法流程，并在合适时机进行不同种群个体的交流。在本文，我们将讨论不同并行策略对遗传算法的加速效果以及求解效率。

关键字：TSP 问题；遗传算法；并行化；

Abstract The traveling salesman problem is an NP hard problem in combinatorial optimization and is very important in operations research and theoretical computer science. As the scale of the problem increases, the precise algorithm becomes powerless to solve the problem. Therefore, scholars at home and abroad focus on the approximate algorithm or heuristic algorithm to solve this problem. Genetic algorithm (GA), as a heuristic algorithm, can quickly search out all the solutions in the solution space and has a strong global search ability, which overcomes the problem of accelerating descent trap of other algorithms. As the scale and complexity of the problem increases, it takes longer for the serialization genetic algorithm to converge to the global optimum. By virtue of the natural parallelism of genetic algorithm, we

adopt different parallel strategies to accelerate the algorithm to solve the global optimal value. The difficulty of genetic algorithm lies in how to ensure the diversity of population while keeping the excellent individuals. Therefore, the most commonly used parallel strategy is to initiate multi-population simultaneous algorithm flow, and at the appropriate time for the exchange of different population individuals.In this paper, we will discuss the accelerating effect and solving efficiency of different parallel strategies on genetic algorithm.

Keywords: Travelling salesman problem; genetic algorithm; parallel strategy

一、介绍

旅行商问题（TSP）[1]是一个经典的组合优化问题，其可以描述为：一个商品推销员要去若干个城市推销商品，该推销员从一个城市出发，需要经过所有城市后，回到出发地。我们应如何选择行进路线，以使总的行程最短。从图论的角度来看，该问题实质是在一个带权完全无向图中，找一个权值最小的 Hamilton 回路。由于该问题的可行解是所有顶点的全排列，随着顶点数的增加，会产生组合爆炸，因而它是一个 NP 完全问题[2]。由于其在交通运输、电路板线路设计以及物流配送等领域内有着广泛的应用，国内外学者对其进行了大量的研究。早期的研究者使用精确算法求解该问题，但随着问题规模的增大，精确算法将变得无能为力。因此，在后来的研究中，国内外学者重点使用近似算法或启发式算法，主要有遗传算法[4]、模拟退火法[3]、蚁群算法等。

本文主要研究遗传算法求解 TSP 问题的实现与并行化策略。遗传算法的训练不仅通过单个生物体的适应来完成，还通过种群的进化来实现。该方法从代表问题中生成一个或多个初代种群（解集），解集中的解又被称为个体，而个体的本质是带有经过基因编码的特征的实体。经过适应函数筛选的个体形成的新的种群，遗传算法不断地评价每一个个体，保证更适应环境的个体拥有更多的繁殖机会。遗传算法重视的是种群之间的搜索策略，以及个体信息在种群内的交换，克服了传统搜索算法难以解决非线性复杂问题的缺点，具有适合并行处理、鲁棒性强、简单通用、搜索能力强和运用范围广的特点[5]。

但随着问题的规模和复杂程度不断增大，遗传算法收敛到全局最优所花费的时间也更长。而且普通遗传算法在进化后期搜索效率较低，存在收敛过早的风险[7]。因此，我们希望优化算法的流程，在充分保留优良个体的同时维持群体的多样性。考虑到遗传算法具备天然的并行性，我们可以从种群的角度来设计并行化遗传算法[8]。现在主要包括四种基本模型：主从模型、孤岛模型、邻域模型和混合模型。主从模型是一种较为直接的并行化策略，其将适应度评价分配至从节点，而种群的选择、交叉、变异在主节点完成[9]。孤岛模型又称粗粒度模型，将多个种群分配至各个节点并各自运行遗传算法[10]。在经过一定代数后，子群体交换优良个体，丰富了子群体的多样性，降低了过早收敛的可能。而混合模型则是将前面的几种模型进行优势的互补，其中效果较好的组合方式是：主从式-粗粒度模型[11]。本文主要讨论普通遗传算法的实现以及对其进行主从式、粗粒度模式的并行优化。当然，我们为了更深入了解粗粒度模型还讨论其交换不同种群个体的策略，以尽可能加速算法的同时保证收敛速度较快。

二、串行算法

1. 遗传算法

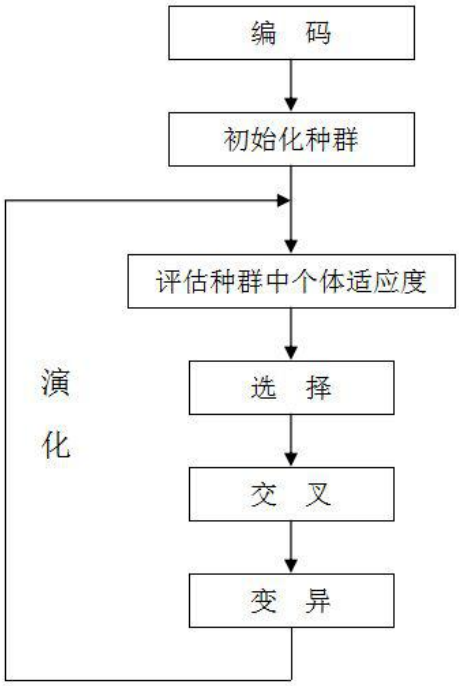

图 1 遗传算法流程图

遗传算法是从代表问题可能潜在的解集的一个种群开始运算。而一个种群则

由经过基因编码的一定数目的个体（解）组成，且每个个体实际上是染色体带有特征的实体。针对不同的问题，个体编码的方式是往往是不同的。染色体作为遗传物质的主要载体，即多个基因的集合，其内部表现（即基因型）是某种基因组合，它决定了个体的形状的外部表现，如黑头发的特征是由染色体中控制这一特征的某种基因组合决定的。因此，在一开始需要实现从表现型到基因型的映射即编码工作。由于仿照基因编码的工作很复杂，我们往往进行简化，如二进制编码法、符号编码法等等。

待初代种群产生后，根据达尔文进化论的指导，我们按照适者生存和优胜劣汰的原理，逐代演化产生出越来越好的近似解，在每一代，根据问题域中个体的适应度大小选择个体，并借助于自然遗传学的遗传算子进行组合交叉和变异，产生出代表新的解集的种群。以上过程将导致种群像自然进化一样的后生代种群比前代更加适应于环境，末代种群中的最优个体经过解码，可以作为问题近似最优解。下面我们主要介绍算法中的关键步骤，并讨论各步骤并行化的可能性。

1.1 编码

在对不同问题建模的时候，我们需要设计相应的编码方式将场景涉及的输入转换为遗传算法的输入。现在广泛采用的是三种编码方式：二进制编码法、符号编码法以及浮点数编码法。

针对求解函数极值等问题时，我们一般采用二进制编码法。该类方法通过足够长的二进制染色体编码使得编码、解码、交叉、变异等操作更加简单。然而该法存在着连续函数离散化时的映射误差。个体长度较短时，可能达不到精度要求，而个体编码长度较长时，虽然能提高精度，但增加了解码的难度，使遗传算法的搜索空间急剧扩大。

针对本文的 TSP 问题，我们采用符号编码的方式对输入进行建模。该编码方式是指个体染色体编码串中的基因值取自一个无数值含义、而只有代码含义的符号集。例如，我们考虑一个 3 城市的 TSP 问题，那么每个城市被赋予一个唯一编号：{1,2,3}。初始化的编码种群就为：{1-2-3, 1-3-2, 2-1-3, 2-3-1}，每个个体分别代表一种问题解的路径。

1.2 适应度评价

为了寻求问题的最优解，我们需要一个评估个体优劣程度的指标——适应度函数（评价）函数。适应度函数总是非负的，而目标函数可能有正有负，故需要在目标函数与适应度函数之间进行变换。评价个体适应度的一般过程为：

1. 对个体编码串进行解码处理后，可以得到个体的表现型。

2. 由个体的表现型可计算出个体的目标函数值。

3. 根据最优化问题的类型，由目标函数值按一定的转换规则求出个体的适应度。

在求解 TSP 问题时，我们根据个体解码后得到的路径来计算长度。本文采用的适应度函数为： $\text{fitness} = \exp(\frac{2\times N}{\text{distance}})$，其中 N 表示城市的数目，而 distance 表示解路径的欧式距离长度。考虑到计算不同个体的适应度，主从式模型将该部分操作分配至从属机。如此一来提高计算效率，加速了个体适应度的计算过程。

1.3 选择

选择运算把当前群体中适应度较高的个体按某种规则遗传到下一代种群中。一般要求适应度较高的个体将有更多的机会遗传到下一代。我们在本文使用轮盘赌选择算子，这是一种回放式随机采样方法。每个个体进入下一代的概率等于它的适应度值与整个种群中个体适应度值和的比例。

1.4 交叉

交叉运算是遗传算法中产生新个体的主要操作过程，它以某一概率相互交换某两个个体之间的部分染色体。其具体操作过程为：1. 先对种群进行随机配对；2. 随机设置交叉点位置；3. 相互交换配对染色体之间的部分基因。以 TSP 问题为例，我们考虑两个双亲个体 1-2-3-4 与 4-3-1-2，其中红色部分为选定交叉点。经过交换后产生的子代个体分别为：2-3-1-4 和 4-2-3-1。

1.5 变异

变异运算是对个体的某一个或某一些基因序列上的基因值按某一较小的概率进行改变。这也是一种产生新个体的操作方法。我们采用基本位变异的方法来进行变异运算，其具体操作过程为：1. 首先确定出各个个体的基因变异位置，其中的数字表示变异点设置在该基因处；2. 然后依照某一概率将变异点的原有基因值取反。在 TSP 问题中，我们假设某个个体基因型为：1-2-3，若红色处被

选中为突变位置，则基因型改变为：2-1-3 或者 1-3-2。

2. 串行优化策略

由上述遗传算法的流程，我们可以看出：编码初始种群的操作仅仅涉及一次运算，而实际影响算法求解的迭代步骤为：适应度评价、选择、交叉和变异。因此，针对这四个步骤进行并发优化可以取得一定的加速效果。

针对适应度评价过程，我们需要先解码每个个体的基因型后得到表现型，然后根据适应度函数计算出适应度值。这个过程可以分配至多个线程并发计算每个个体的适应度，做到更细粒度的算法加速。同样对于交叉运算和变异运算，我们同样可以按照上面的做法进行并发处理。

在后面的实验对比中，我们会将串行算法与适应度评价、选择、交叉、变异等操作并发处理的策略进行比较。这种细粒度的并发操作应该会一定程度上加速算法的运行效率。

三、并行策略

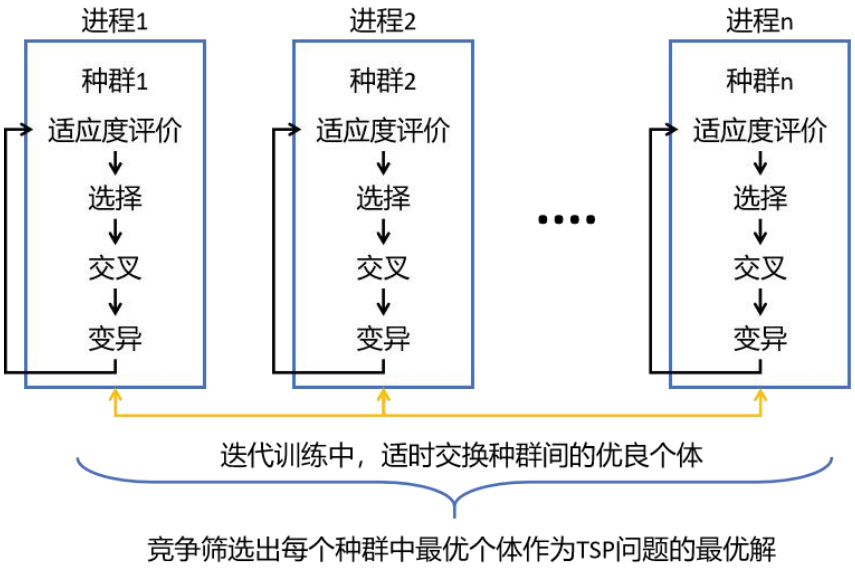

图 2 并行化遗传算法

前面我们介绍了对串行算法的关键操作进行了并发优化设计。但普通遗传算法的缺点依然存在，即：随着问题规模的增大，遗传算法后期搜索效率较低，收敛到全局最优所花费的时间也更长。因此，我们希望改进后的算法能在充分保留优良个体（解）的同时，又要维持种群的多样性。这也是在复杂解空间中寻找最

优解的启发式设计。考虑到遗传算法天然的并行性，我们针对遗传算法的特点进行相应的并行化设计。目前遗传算法的并行化模型共分为四类：主从式模型、粗粒度模型、细粒度模型和混合式模型。我们本文只讨论主从式模型和粗粒度模型。

1. 主从式模型：该方法是一种直接并行化设计的方案，主要解决个体适应度评估操作时计算量较大的情况。该方案只考虑一个种群的情况，种群的选择、交叉、变异等操作都在主机上完成，而从机接收发来的个体进行适应度值计算。待从机计算完成后，主机接收从机的计算结果后继续进行后续流程。

2. 粗粒度模型：粗粒度模型又称孤岛模型，是一种保证种群多样性同时加速计算的较优方案。我们依照进程数量或节点机数量初始化多个种群（子种群），然后各子群体在所处的节点或进程上运行遗传算法。待经历一定进化代数后，子群体交换部分优良个体，以保证种群的多样性，降低未成熟就过早收敛的可能。

以上模型根据其自身特点来说，各有优势。在本文中，我们采用粗粒度模型对串行遗传算法进行加速。由于前文介绍了串行优化中，我们针对个体适应度评价采用了并发编程技术。所以，我们不再使用混合模型的思想来结合主从式模型和粗粒度模型的优点。

并行化遗传算法示意图如图 2 所示。我们在本机实验中，开启多个进程后初始化不同子种群，并开始各自的遗传算法流程。每经过一定世代数后，我们选取每个种群中的优良个体进行交换。本文会讨论该步骤中交换的个体数对算法的收敛性和加速效果的影响。最后待算法满足结束条件后，我们选取种群中的最优个体作为问题最优解。

四、算法设计与实现

本部分针对遗传算法中各个算子的设计实现以及并行化流程进行详细介绍。

4.1 适应度评价

我们根据个体解码后得到的路线来计算总路径长度。本文采用的适应度函数为：$\text{fitness} = \exp(\frac{2\times N}{\text{distance}})$，其中 N 表示城市的数目，而 distance 表示解路径的欧式距离长度。得到每个个体的适应度值之后，我们再利用该值进行选择运算操作。具体评价函数的实现如下：

```
def get_fitness(self, line_x, line_y):
    '''''
        Get the fitness values according to minimizing the distances between cities
    '''
    total_distances = np.zeros((line_x.shape[0], ), dtype=np.float64)
    for i, (xs, ys) in enumerate(zip(line_x, line_y)):
        total_distances[i] = np.sum(np.sqrt(np.diff(xs)**2 + np.diff(ys)**2))
    fitness = np.exp(2. * self.DNA_size / total_distances)
    return fitness, total_distances
```

4.2 选择运算

我们进行选择运算的依据是轮盘赌算法。根据上一步骤计算得到的适应度值，我们计算所有个体的适应度之和。然后，每个个体的适应度与总适应度的比值即为该个体被保留下来的概率。因此，适应度越高的个体被保存下来的几率越大。具体算法实现如下：

```
def select(self, fitness, pop):
    '''''
        Select the DNA to the next generation accoding to the fitness value
    '''
    size = len(pop)
    idx = np.random.choice(np.arange(size), size=size,
                           replace=True, p=fitness/fitness.sum())
    return pop[idx]
```

4.3 变异运算

我们对个体中每个位置的基因点进行变异运算。当该位置的采样概率小于变异概率值，我们随机选取一个位置点将两者的基因值交换。具体算法实现如下：

```
def mutate(self, child):
    '''''
        Operate the mutation on the DNAs
    '''
    for point in range(self.DNA_size):
        if np.random.rand() < self.mutation_rate:
            swap_point = np.random.randint(0, self.DNA_size)
            child[point], child[swap_point] = child[swap_point], child[point]
    return child
```

4.4 交叉运算

我们以某一概率相互交换某两个个体之间的部分染色体。其具体操作过程为：先对种群进行随机配对；然后随机设置交叉点位置；最后相互交换配对染色体之间的部分基因。具体算法实现如下

```
def crossover(self, parent, pop):
    '''''
        Operate crossover on the DNAs
    '''
    if np.random.rand() < self.cross_rate:
        index = np.random.randint(0, len(pop), size=1)
        cross_points = np.random.randint(0, 2, self.DNA_size).astype(np.bool
)
        keep_cities = parent[~cross_points]
        swap_cities = pop[index, np.isin(pop[index].ravel(), keep_cities, in
vert=True)]
        parent[:] = np.concatenate((keep_cities, swap_cities))
    return parent
```

4.5 并行化算法流程

本文采用的并行化算法主要为：为每个进程分配不同种群，而它们同时进行选择、交叉、变异等运算。待每次完成这些运算后，每个种群的最优个体同步至所有进程的种群。我们根据进程编号为每个进程分配子种群，具体算法实现如下：

```
def split_size(self, size):
    '''''
    Split a size number(int) to sub-size number.
    '''
    splited_idx = {}
    idx = np.arange(size)
    if size < self.size:
        return idx
    else:
        rank_id = 0
        inc = size//self.size
        for i in range(0, size, inc):
            splited_idx[rank_id] = idx[i:] if i + inc >= size\
                                           else idx[i:i+inc]
            rank_id += 1
        return splited_idx[self.rank]
```

待每个种群完成相应的遗传运算后，我们需要收集各个进程的种群最优个体。通过 MPI 的 allgather 函数，我们将本进程的种群发送至其它进程；同时收集其它进程的种群。具体算法实现如下：

```
def merge_seq(self, seq):
    '''''
    Gather data in sub-process to root process.
    '''
    if self.size == 1:
        return seq
    mpi_comm = MPI.COMM_WORLD
    merged_seq = mpi_comm.allgather(seq)
    return merged_seq
```

我们在主进程中进行相关实验结果的输出，包括：运行时间与求解得到的最优值。整个遗传算法的流程函数如下所示：

```
def evolve(self):
    '''''
        Operate the evolutions
    '''
    if self.mpi.is_master:
        start_time = time.time()
    for i in range(self.n_generations):
        local_idx = self.mpi.split_size(len(self.pop))
        pop = self.pop[local_idx]
        line_x, line_y = self.translateDNA(pop)
        fitness, total_distances = self.get_fitness(line_x, line_y)
        pop = self.select(fitness, pop)
        pop_copy = pop.copy()
        for parent in pop:
            child = self.crossover(parent, pop_copy)
            child = self.mutate(child)
            parent[:] = child
        pop = np.vstack(self.mpi.merge_seq(pop))
        self.pop = pop
        if self.mpi.is_master:
            best_index = np.argmax(fitness)
            print(f"No. {i} Generation Total distances {'%.3f'%total_distanc
es[best_index]}")
    if self.mpi.is_master:
        end_time = time.time()
        print(f"Time Consume: {'%.3f'%(end_time - start_time)}s")
```

五、测试结果与性能分析

我们主要讨论不同问题规模下并行算法的加速性能以及不同进程数量对加速性能的影响。同时，为了直观展示遗传算法的求解情况，我们统计了串行与并行场景下的迭代求解最优值的过程。

5.1 进程数为 4，数据规模变化

城市数量	串行(s)	并行(s)	加速比
20	5.140	2.076	2.476
50	6.873	2.856	2.407
100	11.147	4.287	2.600
1000	67.871	34.622	1.960

表 1 城市数量对加速性能的影响

种群数量	串行(s)	并行(s)	加速比
100	1.354	0.734	1.845
500	6.591	3.553	1.855
1000	11.808	5.593	2.111
10000	108.717	56.779	1.915

表 2 种群数量对加速性能的影响

在该部分讨论中，我们将进程数量固定为 4，同时讨论了城市数量和种群数量对并行算法加速性能的影响。如表 1 所示，随着城市数量的增加（问题规模的增大），并行算法的加速性能有先上升后下降的趋势。不过总体而言，并行算法的加速效果明显。而在遗传算法的求解过程中，种群数量也是影响求解最优值的关键因素。如表 2 所示，随着种群数量的增加，并行算法的加速性能同样有先上升后下降的趋势。不过，从加速比的角度来看，种群数量对并行算法性能的影响要略小于城市数量。因此，在设计遗传算法的并行求解方案时，可以考虑不同问题规模的影响，从而设计更高效的求解过程。

5.2 问题规模固定，进程数量变化

进程数量	并行（s）	加速比
1	10.087	1.000
2	4.729	2.357
3	5.138	2.169
4	4.287	2.600

表 3 进程数量对加速性能的影响

在该部分讨论中，我们将问题规模固定为：城市数量 100；种群数量 500，讨论不同的进程数量对并行算法性能的影响。如表 3 所示，并行算法在不同的进程数量下具有不同的加速比。随着进程数量的增加，在保证问题规模划分均匀的情况下，并行化遗传算法的加速性能会有所提升。因此，设计并行算法的一个关键问题是可并行部分的问题划分要被仔细考虑，以免影响最终并行算法的性能。

5.3 问题求解的收敛情况

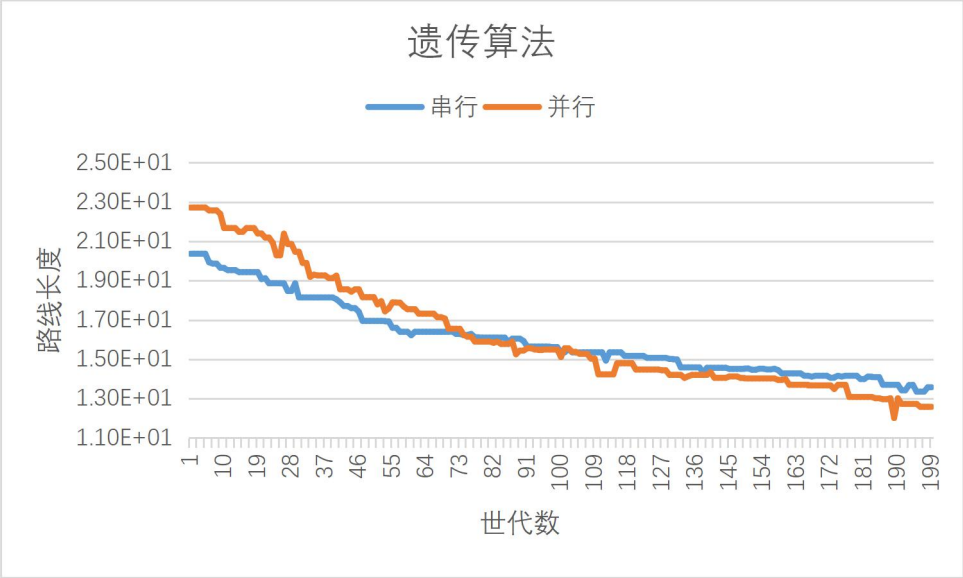

图 3 遗传算法的收敛情况

考虑并行化遗传算法的正确性，我们需要确保问题的求解过程可以收敛。我们将问题规模固定为：城市数量 50；种群数量 500；世代数 200。在每个世代完成相关运算后，我们在主进程统计最优个体所对应的路径之和。如图 3 所示，我们展示了串行算法和并行算法的中间结果。随着迭代次数的增加，并行算法可以很好地收敛至较低的路径长度，表明问题求解的过程是正确的。同时，我们在并行算法中，划分不同的种群至各个进程进行遗传算法，增加了种群的多样性。我们看到：并行化算法求解的最优路径之和要小于串行算法的所求解。这也说明并行遗传算法求解 TSP 问题的优势不仅在于加速性能，还有种群多样性的特点。

六、总结

通过此次并行化遗传算法的设计，我对该启发式算法求解旅行商问题有了更深的认识。本文在研究串行遗传算法的基础上，对选择、交叉、变异等遗传算子进行了并行可行性分析。考虑到不同种群的相关运算存在并行的可能性，我们设

计了主从式结合粗粒度式的并行化遗传算法。结合 MPI 库在 Python 下的第三方支持，我们实现了划分任务以及数据通信等基础函数，帮助实现了并行化遗传算法。

通过分析问题规模以及进程数量对并行化算法的性能影响，我发现子任务的划分对于并行化算法的性能尤为关键。这也为之后设计相关问题的并行求解方案提供了很好的思路。为了直观展示并行算法的收敛性，我统计了迭代求解 TSP 问题的中间结果，证明了算法的正确性。通过整个实验流程，我对遗传算法以及其可优化部分有了更深入的了解。

参考文献

[1] 吴斌, 史忠植. 一种基于蚁群算法的 TSP 问题分段求解算法 [J]. 计算机学报, 2001, 24(12) :1328-1333.

[2] Komusiewicz C, Bulteau L, Hüffner F, et al. Multivariate algorithmics for NP-hard string problems [J]. Bulletin of Eatcs, 2014, 114(114).

[3] Metropolis N, Rosenbluth A W, Rosenbluth M N, et al. Equation of state calculations by fast computing machines [J]. The journal of Chemical Physics, 1953, 21 (6) :1087-1092.

[4] Holland J H. Erratum: generic algorithms and the optimal allocation of trials [J]. Siam Journal on Computing, 2006, 2 (2) :88-105.

[5] Kumar M, Husian M, Upreti N, et al. Genetic algorithm: Review and application [J]. Computer, 2010, 2 (2) :451-454.

[6] Salembier R G, Southerington P. An implementation of the aks primality test [J]. Computer Engineering, 2005.

[7] 宗德才, 王康康. 一种混合局部搜索算法的遗传算法求解旅行商问题 [J]. 计算机应用与软件, 2015, 32 (3) :266-270, 305.

[8] Zeigler B P, Kim J. Asynchronous genetic algorithms on parallel computers [C] //Proceedings of the 5th International Conference on Genetic Algorithms. Morgan Kaufmann Publishers Inc. 1993.

[9] 刘晓平, 安竹林, 郑利平. 基于 MPI 的主从式并行遗传算法框架 [J]. 系统仿真学报, 2004, 16 (9) :1938-1940.

[10] 李志坚, 吴晓军, 任哲坡, 等. 基于分布式粗粒度并行计算的遗传规划算法研究 [J]. 计算机应用研究, 2015, 32 (1) :48-50.

[11] 高家全, 何桂霞. 并行遗传算法研究综述 [J]. 浙江工业大学学报, 2007, 35 (1) :56-59.