Parallel sorting by regular sampling 算法基于Python mpi4py库的实现

21307140069 田沐钊

在本项目中，我们选择用Python与mpi4py库实现Parallel sorting by regular sampling 算法，对结果进行验证，并尝试多种不同的线程数和数据量，分析其加速效果。

代码实现

首先分析PSRS 算法的工作原理如下:

1. 将输入数据分发到多个进程.
2. 每个进程对其本地数据进行排序.
3. 从每个进程的本地排序数据中选择固定数量的采样点.
4. 根进程收集采样点,选择主元值,并将其广播给所有进程.
5. 每个进程根据主元值划分本地数据,并与其他进程交换分区的数据.
6. 每个进程合并收到的分区,并将最终排序的数据发回根进程.
7. 根进程合并所有进程的最终排序数据,得到完整的排序数组.

下面是其代码的具体实现：

```
from mpi4py import MPI
import random
import bisect
import argparse
from itertools import chain


class PSRS():
    def parse_arguments(self):
        parser = argparse.ArgumentParser()
        parser.add_argument('-l', '--length', type=int, required=True, help='Length
of the array')
        return parser.parse_args()

    def data_distribution(self, data, num_proc):
        partition_size = len(data) // num_proc
        partitions = [data[i: i + partition_size] for i in range(0, (num_proc - 1) *
partition_size, partition_size)]
        partitions.append(data[(num_proc - 1) * partition_size:])
        return partitions

    def select_sampling_points(self, data, num_samples):
        return [data[i] for i in range(0, len(data), max(len(data) // num_samples,
1))][:num_samples]
```

```
    def select_pivots(self, samples, num_pivots):
        samples_flattened = sorted(chain(*samples))
        return samples_flattened[num_pivots:len(samples_flattened):num_pivots]
[:num_pivots - 1]

    def partition_local_data(self, local_data, pivot_values):
        partitions = []
        start_index = 0
        for pivot in pivot_values:
            end_index = bisect.bisect_left(local_data, pivot, start_index)
            partitions.append(local_data[start_index:end_index])
            start_index = end_index
        partitions.append(local_data[start_index:])
        return partitions

    def sort_and_merge(self, data_segments):
        return sorted(chain(*data_segments))

    def run(self):
        mpi_comm = MPI.COMM_WORLD
        num_proc = mpi_comm.Get_size()
        rank = mpi_comm.Get_rank()

        # 如果为主进程，则进行数据的生成和分配
        if rank == 0:
            random.seed(666)
            args = self.parse_arguments()
            unsorted_array = [random.randint(0, 10000000) for _ in
range(args.length)]
            # 生成标答，用于最后验证结果
            sorted_array = sorted(unsorted_array)
            start_time = MPI.Wtime()
            data_partitions = self.data_distribution(unsorted_array, num_proc)
        else:
            data_partitions = None

        # 数据被分配给各个进程，进行排序、采样和合并
        local_data = mpi_comm.scatter(data_partitions, root=0)
        local_sorted_data = self.sort_and_merge([local_data])
        sample_points = self.select_sampling_points(local_sorted_data, num_proc)
        gathered_samples = mpi_comm.gather(sample_points, root=0)

        # 在主进程上选择主元
        if rank == 0:
            pivot_values = self.select_pivots(gathered_samples, num_proc)
        else:
            pivot_values = None

        # 将主元广播给所有进程，各进程根据主元划分本地数据，之后合并排序获得最终数据
        pivot_values = mpi_comm.bcast(pivot_values, root=0)
```

```
        local_partitions = self.partition_local_data(local_sorted_data,
pivot_values)
        received_partitions = mpi_comm.alltoall(local_partitions)
        final_sorted_data = self.sort_and_merge(received_partitions)
        final_sorted_data = mpi_comm.gather(final_sorted_data, root=0)

        if rank == 0:
            merged_final_data = self.sort_and_merge(final_sorted_data)
            cost_time = MPI.Wtime() - start_time
            if_correct = (merged_final_data == sorted_array)
            print(cost_time, if_correct)


psrs = PSRS()
psrs.run()
```

下面我们主要分析 PSRS 类的结构:

- `def parse_arguments(self)`: 这个方法创建了一个 `ArgumentParser` 对象,添加了一个数组长度的参数,并返回解析后的参数.
- `def data_distribution(self, data, num_proc)`: 这个方法将输入数据数组划分为等大小的块,分配给每个进程.
- `def select_sampling_points(self, data, num_samples)`: 这个方法从输入数据数组中选择固定数量的采样点.
- `def select_pivots(self, samples, num_pivots)`: 这个方法从采样数据点中选择主元值.
- `def partition_local_data(self, local_data, pivot_values)`: 这个方法根据主元值将本地数据划分.
- `def sort_and_merge(self, data_segments)`: 这个方法将一个列表of排序数据片段合并成一个排序数组.
- `def run(self)`

  : 这个是主方法,协调 PSRS (Parallel Sample Sort) 算法的执行. 它执行以下步骤:

  1. 如果当前进程是根进程(rank 0),它生成一个随机整数数组并将数据分发给所有进程.
  2. 每个进程对其本地数据进行排序,并选择采样点.
  3. 根进程收集所有进程的采样点,选择主元值,并将其广播给所有进程.
  4. 每个进程根据主元值划分本地数据,并与其他进程交换分区的数据.
  5. 每个进程合并收到的分区,并将最终排序的数据发回根进程.
  6. 根进程合并所有进程的最终排序数据,并检查结果是否正确.

这个程序可以通过以下命令行调用运行:

```
mpiexec -n {procs} python psrs.py -l {length}
```

这里计算加速比时，出于便捷考虑，直接将线程为1的运行时间作为串行时间。并且在计算运行时间时，选择每种条件运行10次程序取平均，这样得到的数据会更加可靠。

实验结果与分析

数据量为 1000

线程数	1	2	4	8	16
运行时间（ms）	0.411	1.327	1.733	2.224	5.51
加速比	1.0	0.31	0.23	0.185	0.075

数据量为 10000

线程数	1	2	4	8	16
运行时间（ms）	2.61	2.94	3.20	3.95	6.51
加速比	1.0	0.89	0.82	0.66	0.401

数据量为 100000

线程数	1	2	4	8	16
运行时间（ms）	33.38	23.69	17.09	16.21	25.06
加速比	1.0	1.41	1.95	2.05	1.33

数据量为 1000000

线程数	1	2	4	8	16
运行时间（ms）	577.47	348.21	252.42	202.38	192.12
加速比	1.0	1.66	2.287	2.85	3.005

可见PSRS 算法的加速效果与数据量和线程数有密切关系:

1. 对于较小的数据量, PSRS 算法的并行性收益有限, 加速比随线程数增加而下降.
2. 对于较大的数据量, PSRS 算法能够充分利用并行处理的优势, 加速比随线程数增加而提高.
3. 存在一个最佳线程数, 超过这个数量, 加速比会开始下降, 体现了 CPU 资源利用效率的瓶颈.

我们对其进行分析和猜想如下:

1. **任务粒度和并行度**:
    - PSRS 算法有三个主要步骤:分割、排序和合并.
    - 当数据量较小时,分割和合并的开销相对较大,限制了并行度的提升效果.

- 随着数据量增大,每个子任务的计算工作量也相应增大,并行度的优势就能够更好地发挥.

2. **内存访问和缓存命中率**:
   - 对于较小的数据量,每个线程处理的数据量较少,可以充分利用 CPU 缓存,缓存命中率较高.
   - 随着数据量增大,每个线程需要处理的数据量也随之增大,缓存命中率下降,内存访问开销增加,降低了并行加速的效果.
3. **线程调度和同步开销**:
   - 当线程数较少时,线程调度和同步开销相对较小.
   - 随着线程数增加,线程间的协调和同步开销也会变大,降低了并行处理的收益.
4. **计算密集型和内存密集型**:
   - PSRS 算法的主要计算工作是排序,属于计算密集型.
   - 当数据量较小时,排序计算占主导,并行化效果较好.
   - 当数据量较大时,内存访问开销变得更加重要,限制了并行化的效果.

结论

1. 对于较小的数据量, PSRS 算法的并行性收益有限, 加速比随线程数增加而下降.
2. 对于较大的数据量, PSRS 算法能够充分利用并行处理的优势, 加速比随线程数增加而提高.
3. 存在一个最佳线程数, 超过这个数量, 加速比会开始下降, 体现了 CPU 资源利用效率的瓶颈.