并行与分布式作业

Foster并行程序设计方法

第四次作业

姓名：谷正阳

班级：行政一班

学号：18308045

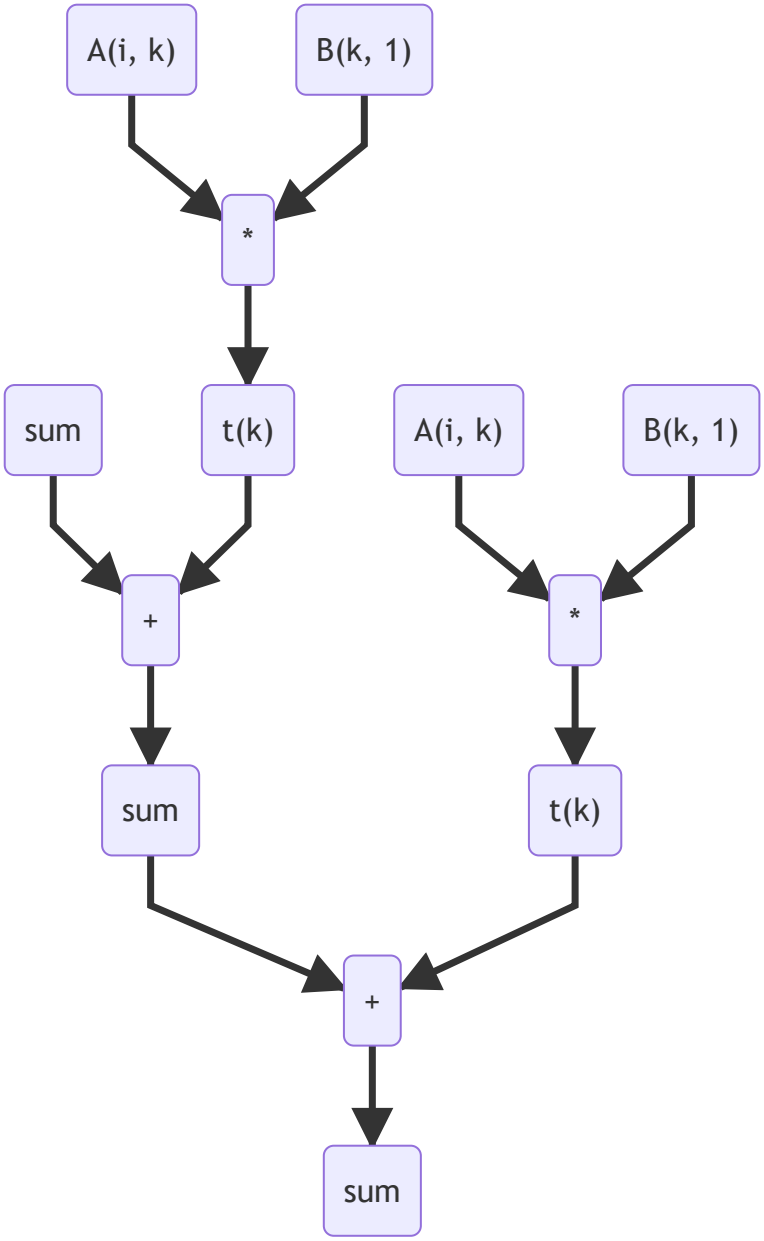
一、问题描述

> 利用Foster并行程序设计方法计算1000x1000的矩阵与1000x1的向量之间的乘积, 要求清晰地呈现 Foster 并行程序设计的四个步骤。

二、解决方案

1. Partitioning

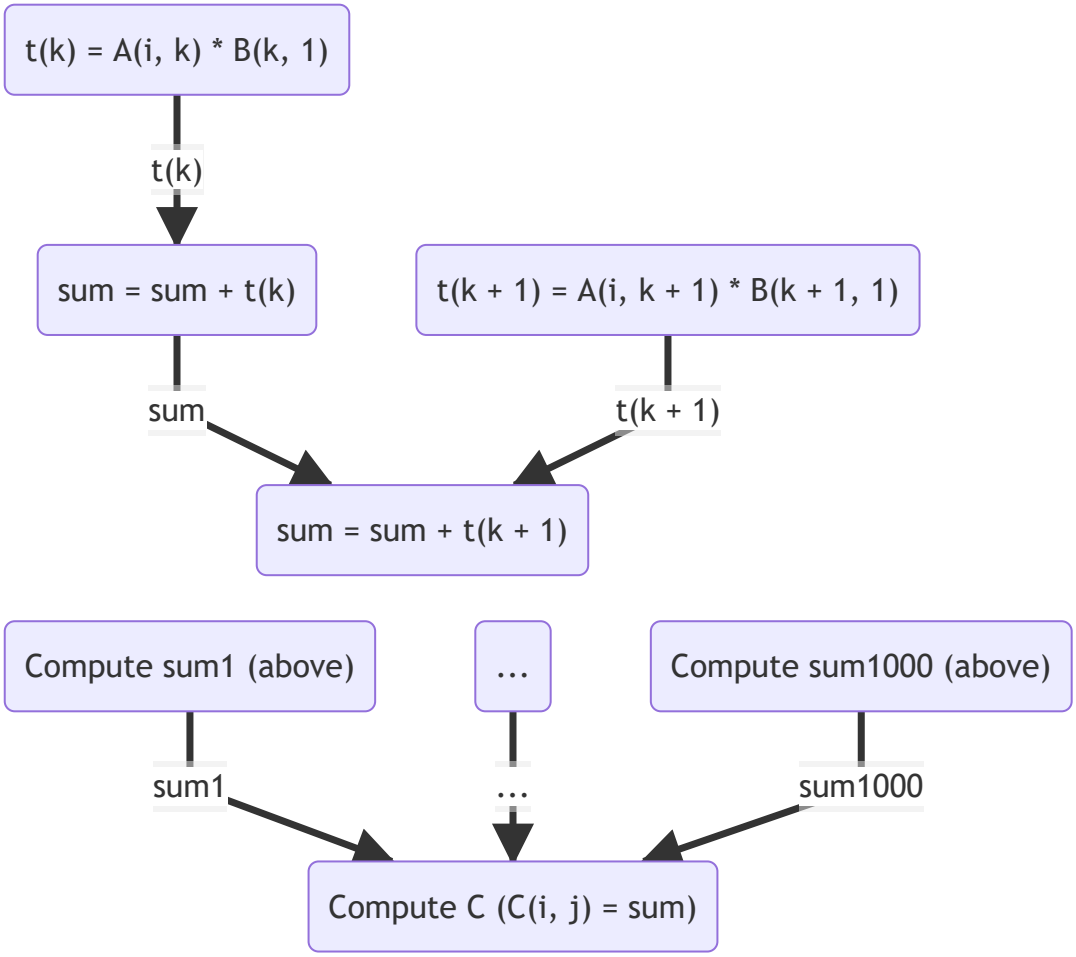
> ∵ 要得到$C = A \cdot B$，需要对于$i, k \in Z$且$1 \leq i, k \leq 1000$有$c_{i,1} = \sum_{k=1}^{1000}(a_{i,k} \cdot b_{k,1})$
> ∴ 按照数据划分，计算C的每行即每个$c_{i,0}$为一份，且相互之间没有依赖关系
> ∴*Exploit data parallelism*
> ∴ 按照任务划分，乘法运算和求和运算
> 设$t_0 = 0, t_k = a_{i,k} \cdot b_{k,1}$，则$c_{i,0} = \sum_{i=0}^{1000}(ti)$
> ∴ 乘法和求和有依赖关系，要先算出$t_k = a_{i,k} \cdot b_{k,1}$再算出$\sum_{i=0}^{k-1} t_i + t_k$
> ∴ *Exploit pipeline parallelism*，对于$2 \leq k \leq 1000$，在计算$a_{i,k} \cdot b_{k,1}$的同时计算$\sum_{i=0}^{k-2} t_i + t_{k-1}$
> ∴ *Dependence graph*:



2. Communication

∵ *Local communication*

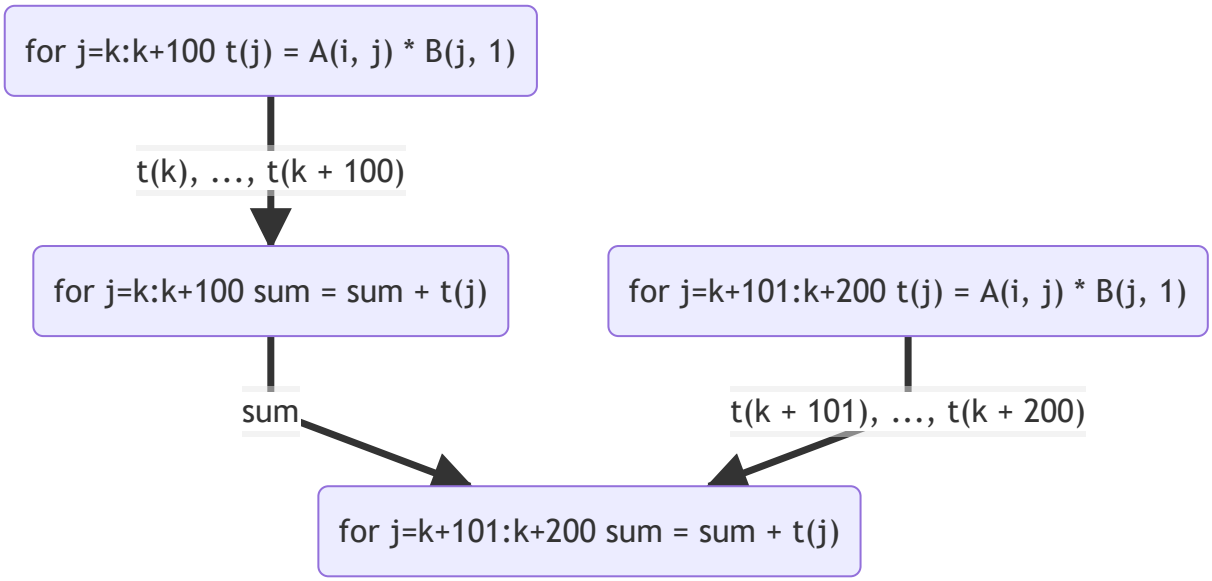
∴ *Task-channel graph*:



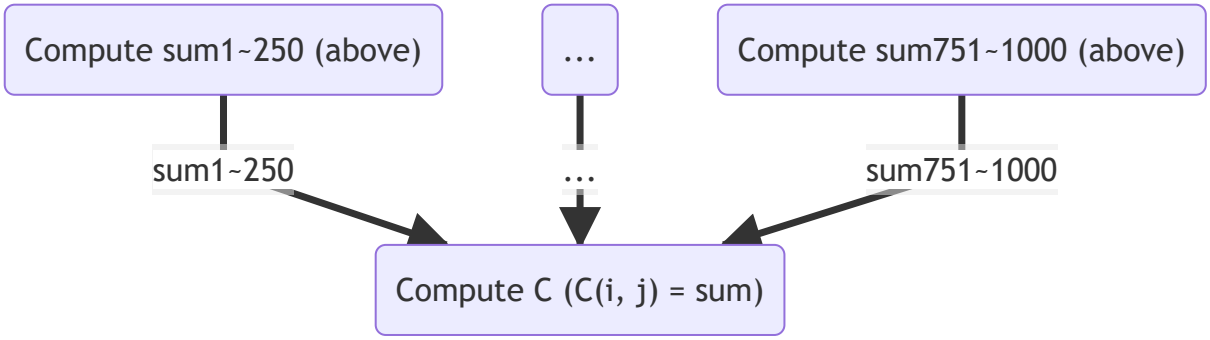
3. Agglomeration

∵ *Combine groups of sending and recieving tasks*

∴ *Task-channel graph*:



∴ *Task-channel graph*:



4. Mapping

> ∵ *Static number of tasks*且*Constant computation time per task*
>
> ∴ *Create one task per processor*

```
int step0 = row / 4;
int stt2 = 2 * step0;
int stt3 = 3 * step0;
int step1 = row_col / 10;
auto start = steady_clock::now();
thread t1(mat_mul, 1, step0, step0, step1);
thread t2(mat_mul, 2, stt2, step0, step1);
thread t3(mat_mul, 3, stt3, step0, step1);
mat_mul(0, 0, step0, step1);
t1.join();
t2.join();
t3.join();
auto end = steady_clock::now();
```

```
mul(t, i, 0, step1);
for (k = step1; k < row_col; k += step1)
{
   thread t1(mul, t, i, k, step1);
   sum(t, i, k - step1, step1);
   t1.join();
}
sum(t, i, k - step1, step1);
```

三、实验结果

> 设 `row` 为矩阵 `A` 及矩阵 `C` 的行数， `row_col` 为矩阵 `A` 的列数，矩阵 `B` 的行数， `col` 为矩阵 `B` 及 `C` 的列树。 `A` ， `B` 初始化为全1， `C` 初始化为全0，求 `C = A * B` 。则预测 `C` 为每个元素都为 `row_col` 。使用 `bool verify()` 检查结果是否正确。

```
bool verify()
{
    for (int i = 0; i < row; i++)
    {
        if (C[i][0] - row_col >= 1 || C[i][0] - row_col <= -1)
        {
            return false;
        }
    }
    return true;
}
```

第一个是用数据并行和流水线，第二个是只有数据并行，第三个是只有流水线，第四个是不并行。

```
verify: 1
time:   2660929024.000000

verify: 1
time:   3543800.000000

verify: 1
time:   4918962176.000000

verify: 1
time:   4605600.000000
```

结果是流水线会拖慢速度，考虑创建线程占用时间太长。如果增加 `row_col` 相比不适用流水线可能会性能提升。如下是1000x10000的矩阵和10000x1的矩阵的结果。

```
verify: 1
time:   2673693952.000000

verify: 1
time:   22821500.000000

verify: 1
time:   5002361344.000000

verify: 1
time:   49414900.000000
```

结果是使用流水线的时间前后基本没变，没使用流水线则前后差了一个数量级。大约在 `row_col` 为10000000时流水线性能超过无流水线。下面是4x10000000和10000000x1矩阵乘法结果。

```
verify: 1
time:   142800496.000000

verify: 1
time:   155319808.000000

verify: 1
time:   165574896.000000

verify: 1
time:   184499200.000000
```

使用流水线性能已经超过不用流水线。

四、遇到的问题及解决方法

问题1. *Dependence graph*和*Task-channel graph*的绘制。
解决1. 使用markdown自带的mermaid实现。

```
graph TD
A0("Compute sum1 (above)")
A1("...")
A3("Compute sum1000 (above)")
B("Compute C (C(i, j) = sum)")
A0=="sum1"==>B
A1=="..."==>B
A3=="sum1000"==>B
```

问题2. 流水线的设计。
解决2. 该步求和和乘法要并发，且该步和下步要满足一个同步关系。使用 `thread` 多线程并发， `join()` 方法实现同步关系。

```
thread t1(mul, t, i, k, step1);
sum(t, i, k - step1, step1);
t1.join();
```

问题3. 计算结果正确性的验证。
解决3. 限定两个矩阵都是全1矩阵，则结果可以通过 `bool verify()` 验证。

问题4. 计算性能的评价。
解决4. 类似实验一，通过 `chrono` 库提供的方法较为精确地计算运行时间。