## 算法简介

将矩阵 $A$ 和 $B$ 分成 $p$ 个方块 $A_{i,j}$ 和 $B_{i,j}(0\leqslant i,j\leqslant\sqrt{p}-1)$，

每块大小为 $(n/\sqrt{p})\times(n/\sqrt{p})$，并将它们分配给 $\sqrt{p}\times\sqrt{p}$ 个处理器 $(P_{0,0},P_{0,1},\cdots,P_{\sqrt{p}-1,\sqrt{p}-1})$。

开始时处理器 $P_{i,j}$ 存放有块 $A_{i,j}$ 和块 $B_{i,j}$，并负责计算块 $C_{i,j}$，然后算法开始执行：

① 将块 $A_{i,j}(0\leqslant i,j<\sqrt{p})$ 向左循环移动 $i$ 步；

将块 $B_{i,j}(0\leqslant i,j<\sqrt{p})$ 向上循环移动 $j$ 步；

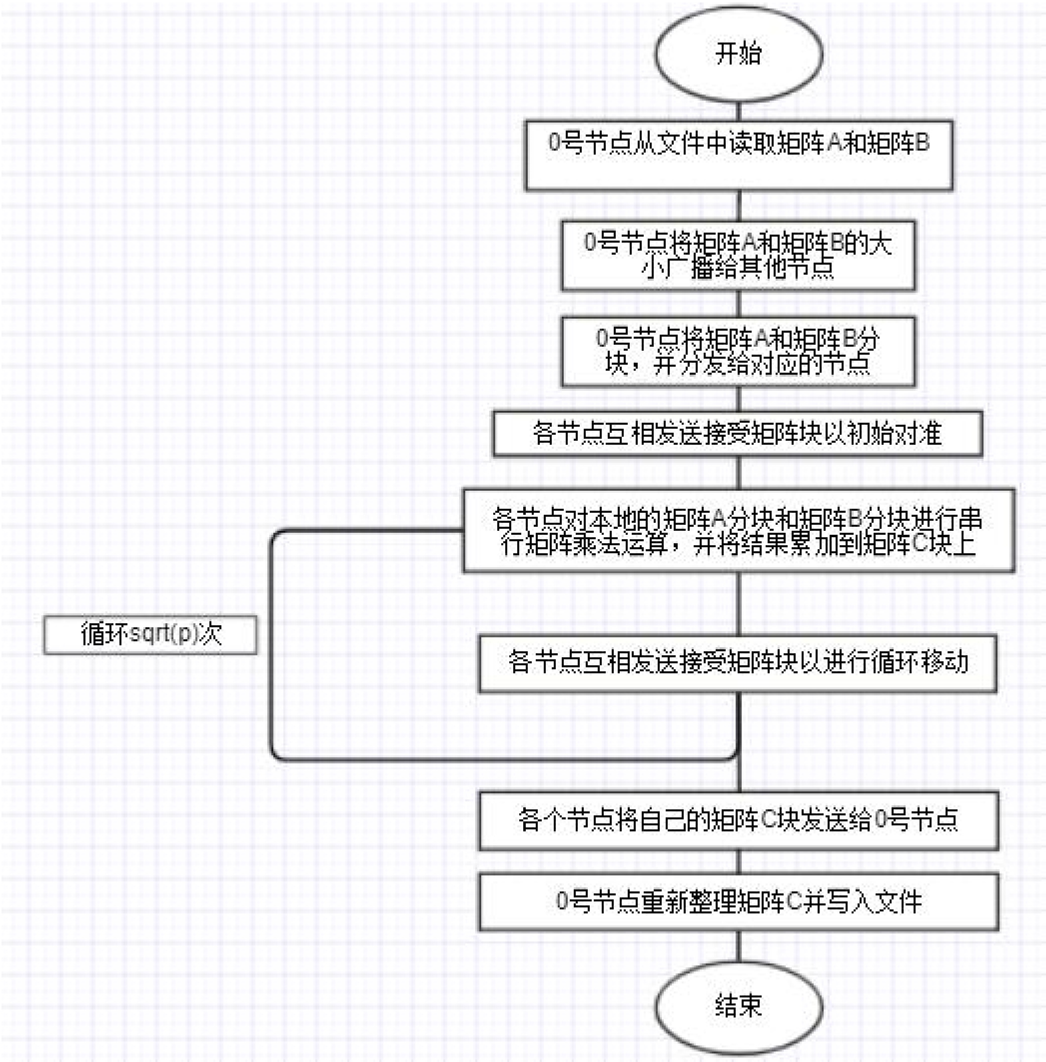
② $P_{i,j}$ 执行乘－加运算；

然后，将块 $A_{i,j}(0\leqslant i,j<\sqrt{p})$ 向左循环移动 1 步；

将块 $B_{i,j}(0\leqslant i,j<\sqrt{p})$ 向上循环移动 1 步；

③ 重复第②步，在 $P_{i,j}$ 中共执行 $\sqrt{p}$ 次乘－加运算和 $\sqrt{p}$ 次块 $A_{i,j}$ 和 $B_{i,j}$ 的循环 单步移位。

## 算法流程图




## 算法设计方法和模式

任务划分：

根据矩阵乘法公式中的累加计算的可分离性，将参与计算的两个矩阵分解成 p 个小矩阵块(共有 p 个计算节点)，每个节点只进行局部的小矩阵乘法，最终计算结束后将局部的小结果矩阵发送回 Master 节点。

$$c_{12} = \sum_{k=0}^{n-1} a_{1k} b_{k2} = a_{10} b_{02} + a_{11} b_{12} + a_{12} b_{22} + \boxed{a_{13} b_{32}}$$

通讯分析：

由于算法在下发任务和收集结果的时候采用了主从模式，所以使用了 Master-Worker 的全局通讯，该部分通讯由于发送方只有一个 0 号线程，所以无法并行执行，只能串行执行。同时，在迭代进行小矩阵运算时，各计算节点之间也需要交换矩阵，进行了结构化通讯。该部分通讯由于通讯的局部特性，可以并行执行，能够提高效率。

任务组合：

每个节点负责一个小矩阵的串行计算，同时负责小矩阵之间的通讯传递。

处理器映射：

由于任务的划分个数等于处理器个数，所以在组合任务的同时完成了处理器映射。

Cannon 算法采用了主从模式的同时也采用了分而治之的模式。一方面，0 号线程作为 Master，负责矩阵 A 和矩阵 B 以及矩阵 C 的 I/O，也负责小矩阵的分发和结果的聚集。而其他节点作为 Worker 进行本地的小矩阵串行乘法计算。另一方面，Cannon 算法将两个大矩阵的乘法运算分解为若干各小矩阵的乘法运算，最终计算结束后，将计算结果聚集回来，也采用了分而治之的思想。cannon 算法不仅实现了矩阵乘法运算的并行化，也减少了分块矩阵乘法的局部存储量，节省了节点的内存开销。

## 算法复杂度

设计算的是一个 n*n 的矩阵乘一个 n*n 的矩阵，共有 p 个节点，那么 Cannon 算法的时间复杂度计算如下：

矩阵乘加的时间由于采用了并行化，所以所需时间为：

$$\frac{\mathrm{n}^3}{p}$$

若不考虑节点延迟时间，设节点之间通讯的启动时间为 $\mathrm{t_i}$，传输每个数字的时间为 $\mathrm{t_w}$，则在两个节点间传输一个子矩阵的时间是：

$$\mathrm{t_i} + \frac{n^2 t_w}{p}$$

所以节点之间传输子矩阵所需的时间为：

$$2\sqrt{\mathrm{p}}(\mathrm{t_i} + \frac{n^2 t_w}{p})$$

综上，cannon 算法总的所需时间为：

$$\frac{n^3}{p} + 2\sqrt{p}(t_i + \frac{n^2 t_w}{p})$$

算法的时间复杂度为:

$$O(\frac{n^3}{p})$$

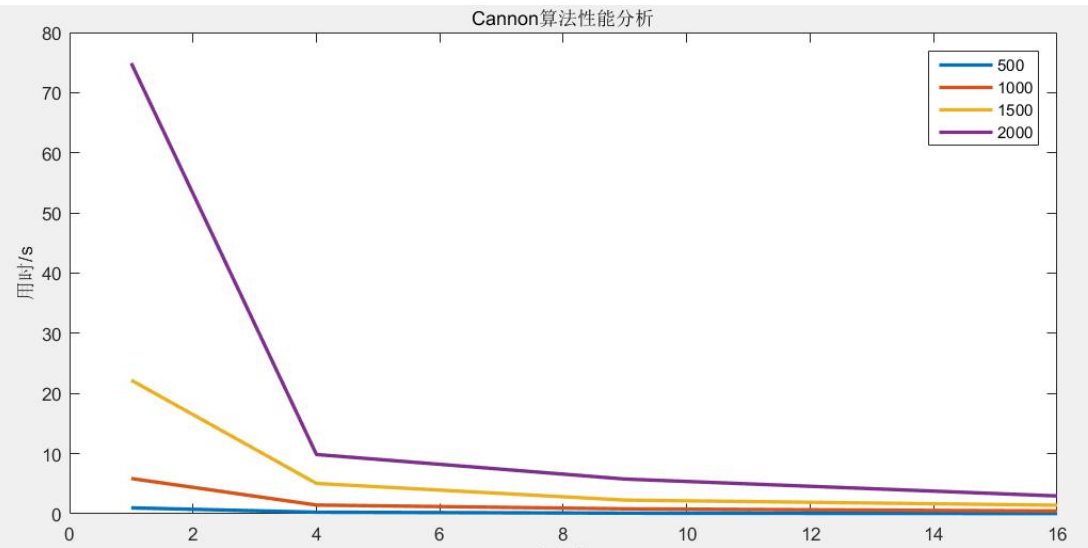
cannon 算法中每个节点（除 0 号 Master 节点外）只需要 3 个小矩阵块大小的内存空间，相比较条带分割的矩阵并行乘法，大大降低了其空间复杂度，共需内存大小 $\frac{3n^2}{p}p = 3n^2$

因此，算法的空间复杂度为：

$$O(n^2)$$

## 算法性能分析

| 矩阵大小 / 线程数 | 500*500 | 1000*1000 | 1500*1500 | 2000*2000 |
|---|---|---|---|---|
| 1 | 0.99 | 5.87 | 22.21 | 74.90 |
| 4 | 0.27 | 1.46 | 5.05 | 9.86 |
| 9 | 0.11 | 0.83 | 2.28 | 5.78 |
| 16 | 0.05 | 0.47 | 1.43 | 2.97 |



除了线程的个数对算法的性能有影响，线程在不同节点的分布也对算法的性能有影响。

经过实验，发现在同一节点下 9 个线程的效率比 3 个节点下，每个节点 3 个线程，共 9 个线程的效率要高，通过 Vampir 分析，发现跨节点的线程通信的代价要远高于本地跨核通信的代价。

这也提醒我们，在设计和运行并行算法时，也与要考虑线程在不同节点之间和本节点之间通信的代价，尽量在本节点内通信。

下面两幅图是进行 1500*1500 的矩阵乘法是，都使用 9 个线程进行计算的性能分析。其中图一使用了 3 个节点，每个节点 3 个线程，图二只使用了 1 个节点，共 9 个线程。可以看出 3 个节点中有两个线程进行线程通信时代价很高，这是因为跨节点通信导致的，而图二则没有这种现象。


图 1

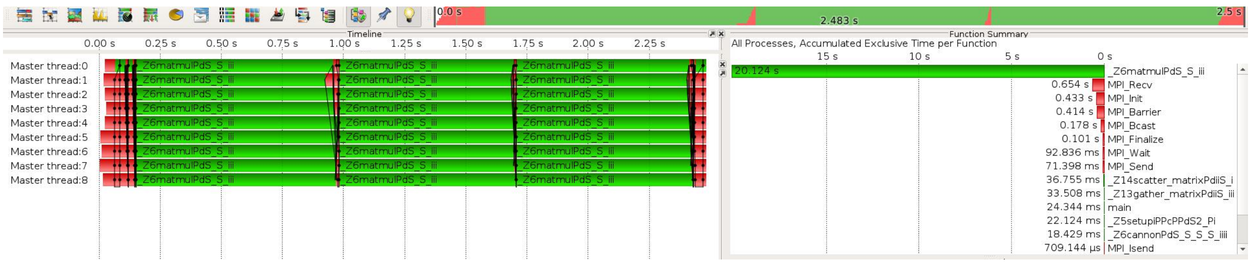



图 2

当矩阵过小时，cannon 算法由于计算时间相比较初始化和通讯代价较小，无法体现出并行算法的优势。

如 11*10 的矩阵乘 10*13 的矩阵，共使用 9 个线程：并行算法和穿行算法苏勇时间都小于 0.01s，如下图 3 所示并行算法中大部分计算量都是初始化工作和线程的通讯。

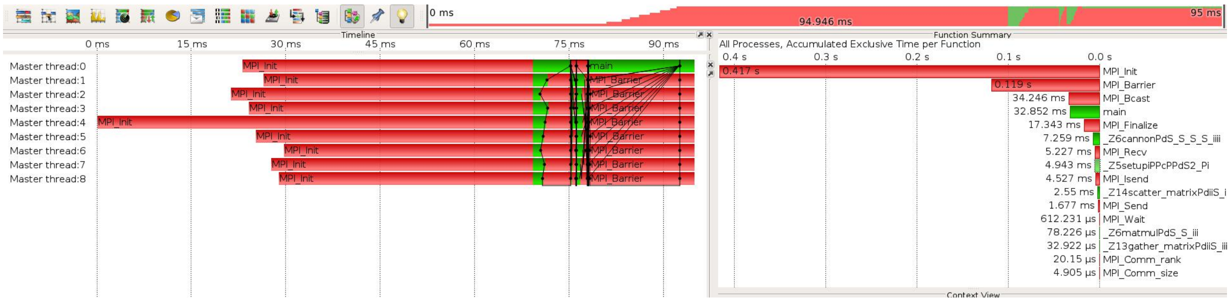


图 3

以 2000*2000 大小的矩阵和 9 个线程的计算为例(图 4)，可以看出，线程出现空闲等待的部分主要包括 0 号 Master 线程分发矩阵和收集结果的步骤，以及每完成一个周期的小矩阵块计算后需要同步再进入下一周期，此期间交换小矩阵块时的通信造成一定程度的空闲等待。

后续可以采用并行读写文件的方式将数据 I/O 也并行化，提高性能。

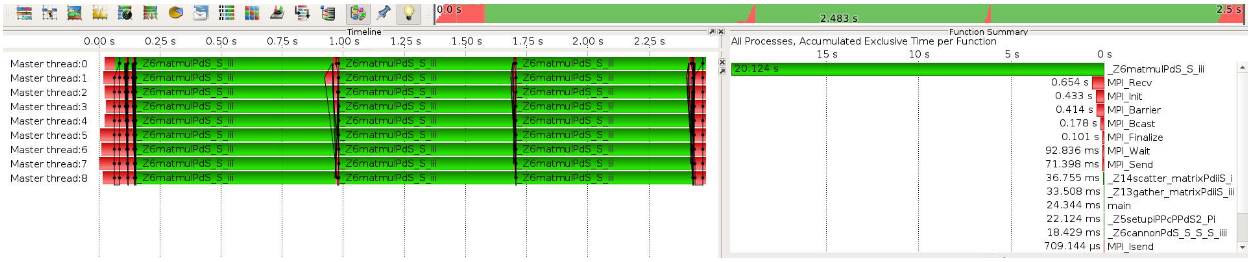


图 4

以下为性能测试的结果数据：

500*500　500*500　1thread 1 node

```
Serial algrithm: multiply a 500x500 with a 500x500, use 0.99(s)
```

500*500　500*500　4thread 1 node

```
Cannon algrithm: multiply a 500x500 with a 500x500, use 0.27(s)
```

500*500　500*500　9thread 1 node

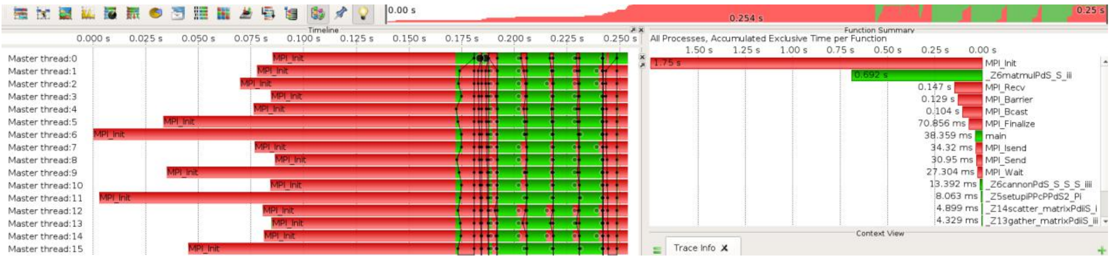
```
Cannon algrithm: multiply a 500x500 with a 500x500, use 0.11(s)
```

500*500　500*500　9thread　3 node

Cannon algrithm: multiply a 500x500 with a 500x500, use 0.15(s)


500*500  500*500  16thread 1 node

Cannon algrithm: multiply a 500x500 with a 500x500, use 0.05(s)




1000*1000  1000*1000  1thread 1 node

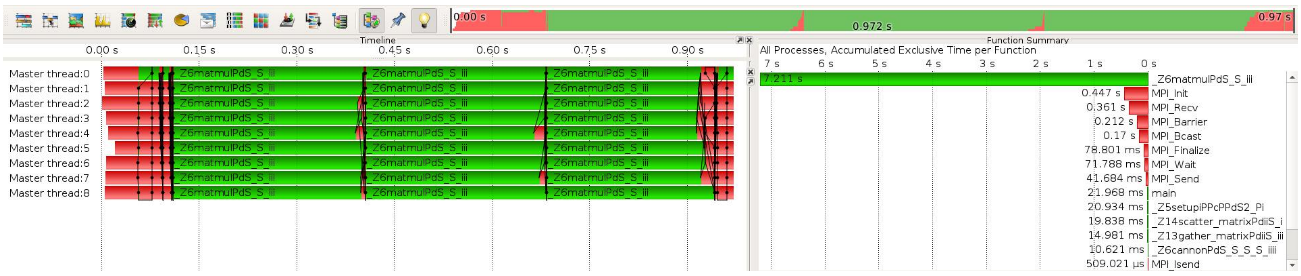
Serial algrithm: multiply a 1000x1000 with a 1000x1000, use 5.87(s)

1000*1000  1000*1000  4thread 1 node

Cannon algrithm: multiply a 1000x1000 with a 1000x1000, use 1.46(s)

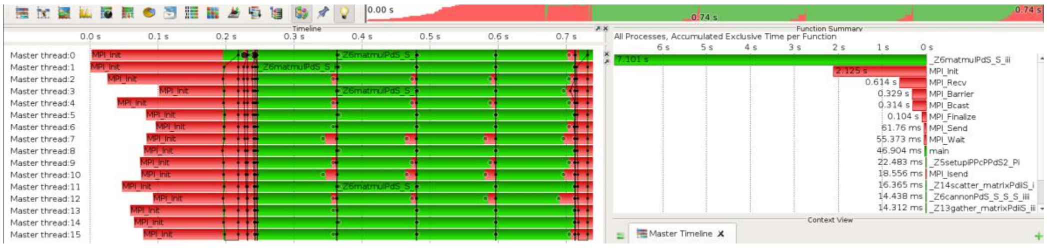
1000*1000  1000*1000  9thread 1 node

Cannon algrithm: multiply a 1000x1000 with a 1000x1000, use 0.83(s)




1000*1000  1000*1000  16thread 1 node

Cannon algrithm: multiply a 1000x1000 with a 1000x1000, use 0.47(s)



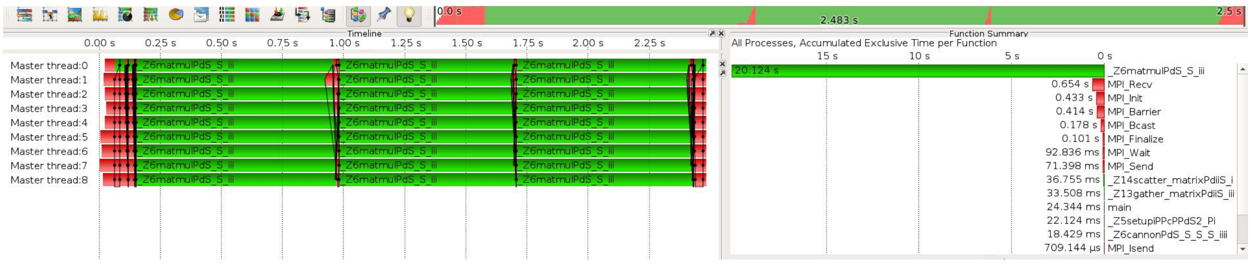
1500*1500  1500*1500  1thread 1 node

Serial algrithm: multiply a 1500x1500 with a 1500x1500, use 22.21(s)

Cannon algrithm: multiply a 1500x1500 with a 1500x1500, use 6.71(s)

1500*1500  1500*1500  9thread 3 node

1500*1500  1500*1500  9thread 1 node



2000*2000  2000*2000  1thread 1 node

Serial algrithm: multiply a 2000x2000 with a 2000x2000, use 77.86(s

2000*2000  2000*2000  9thread 1 node

Cannon algrithm: multiply a 2000x2000 with a 2000x2000, use 23.88(s

2000*2000 2000*2000 9thread 3 node

2000*2000 2000*2000 9thread 1 node