****



**并行计算第二次实验报告**

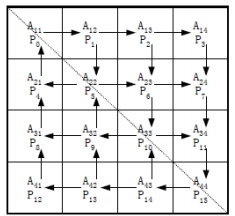
|  |  |
| --- | --- |
| 报告名称： | 多线程实现矩阵转置 |
| 姓 名： | 王红阳 |
| 学 号： | 3019244233 |
| 联系电话： | 15222168550 |
| 电子邮箱： | 1736731090@qq.com |
| 填写日期： | 2021.4.10 |

一、实验内容概述

1. 本次实验要求使用多线程实现矩阵转置算法，主要有两种方法：

（1）块棋盘划分法：

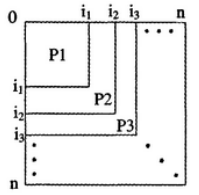
假设处理器个数为 p，编号为 0,1,…,p-1，则将 n 阶矩阵 A 分成 p 个大小为m × m个子块，p 个子块组成一个√p × √p的子块阵列，如图所示：



转置分为两步进行：第一步，子块转置；第二步，处理器内部局部转置。

1. 直角划分方法

直角划分方法如图所示：



转置分为两步进行：第一步，将矩阵划分为大小相近的 p 个子块；第二步，对每一个子块进行转置。

1. 实验要求

实验要求在超算平台上，向超算平台，提交任务需要通过编写 PBS 脚本实现。

本次实验要求使用c语言中的pthread.h函数库，使用多线程实现矩阵转置，并分析实验结果，计算加速比，效率等。

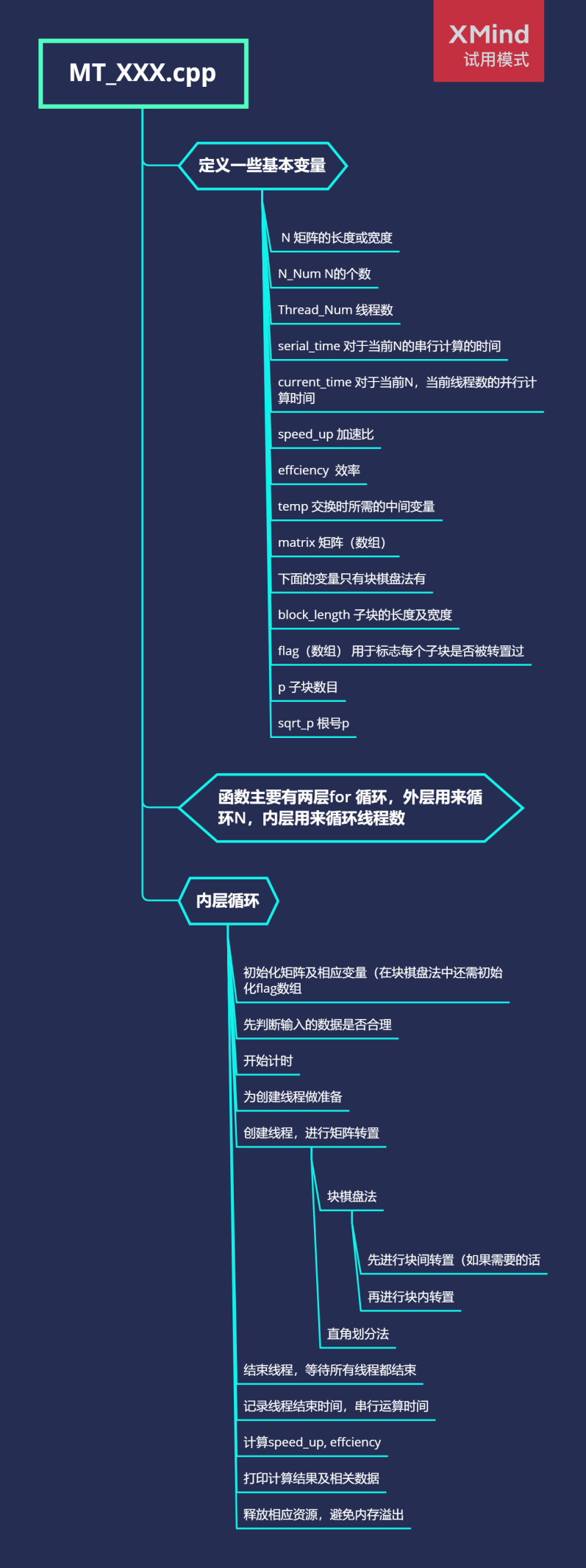
1. 并行计算环境

操作系统：CentOS 7.6

编译环境 ：GCC 4.8.5：g++，

二、算法分析设计

1.代码逻辑图：



注：上图仅针对并行算法的实现，不针对串行算法的实现。

1. 代码原理

直角划分法：

调用头文件pthread.h中的函数，创建相应数量的线程。假设线程数量为t，则可将上述公式中的N分为T份，每份长度不均，每份交给一个线程运行。一般通过id来决定范围，最后所有线程都结束后的最终矩阵即为矩阵转置后的结果。

具体实现如下（传进该函数的变量为进程的序号）：

void \*right\_angle\_division(void \**arg*)

{

    //确定该线程id

    int id = \*(int \*)*arg*;

    //确定线程负责的规模

    int length = N / Thread\_Num;

    //确定线程的截至

    int end = (id == Thread\_Num - 1) ? N : id + length;

    //确定线程的起始

    int begin = id \* length;

    for(int rank=begin;rank<end;rank++)

    {

        for(int i=0;i<rank;i++)

        {

            temp = matrix[rank][i];

            matrix[rank][i] = matrix[i][rank];

            matrix[i][rank] = temp;

        }

    }

    return NULL;

}

块棋盘划分法：

调用头文件pthread.h中的函数，创建相应数量的线程。先根据当前线程号，求出这个线程对应的矩阵块的标号，然后进行矩阵块间的转置（如果每次都进行转置，最终的矩阵相当于没有转置，因为转置与转置相互抵消了，所以需要用flag数组进行标记该矩阵块是否被转置过，若被转置过，则无需转置），然后再进行矩阵块内的转置。

具体实现如下（传进该函数的变量为进程的序号）：

void \*chessboard\_division(void \**arg*)

{

    int rank = \*(int \*)arg;

    //子块在块数组中的横坐标

    int u = rank / sqrt((double)Thread\_Num);

    //子块在块数组中的纵坐标

    int v= rank % (int)sqrt((double)Thread\_Num);

    //块间转置

    if(u == v && flag[u][v] == 0)

        flag[v][v] = 1;

    else if(u != v && flag [u][v] == 0)

    {

        for(int x\_step=0;x\_step<block\_length;x\_step++)

        {

            for(int y\_step = 0;y\_step<block\_length;y\_step++)

            {

                temp = matrix[u\*block\_length+ x\_step][v\*block\_length+y\_step];

                matrix[u\*block\_length+ x\_step][v\*block\_length+y\_step] = matrix[v\*block\_length+ x\_step][u\*block\_length+y\_step];

                matrix[v\*block\_length+ x\_step][u\*block\_length+y\_step] = temp;

            }

        }

        flag[u][v] = flag[v][u] = 1;

    }

    // cout<<"对矩阵块"<<i<<j<<"和矩阵"<<j<<i<<"的交换已完成"<<endl;

    // cout<<"块间转置后的矩阵为："<<endl;

    // print\_matrix();

    //块内转置

    // cout<<"现在正在转置矩阵块("<<u<<","<<v<<")"<<endl;

    for(int x\_step=1;x\_step<block\_length;x\_step++)

    {

        for(int y\_step = 0;y\_step<x\_step;y\_step++)

        {

            temp = matrix[u\*block\_length+ x\_step][v\*block\_length+y\_step];

            matrix[u\*block\_length+ x\_step][v\*block\_length+y\_step] = matrix[u\*block\_length+ y\_step][v\*block\_length+x\_step];

            matrix[u\*block\_length+ y\_step][v\*block\_length+x\_step] = temp;

        }

    }

    return NULL;

}

串行算法

for(int i = 0; i< N;i++)

        {

            for(int j = 0;j<i;j++)

            {

                temp = matrix[i][j];

                matrix[i][j] = matrix[j][i];

                matrix[j][i] = temp;

            }

        }

**三**、编译、运行、测试说明

1. 通过Xshell 6登录超算平台
2. 通过Xftp 6将代码从本级拷贝到该平台的账号上

注：代码编辑器及环境为：VSCODE + WSL

1. 编译代码

经过我的实践发现，PBS脚本在我本次实验中似乎用处不大，只需要在Xshell 6上登录我的超算平台账号，并输入相关编译指令或运行指令即可

生成可执行程序：

命令如下：

g++ -o MT\_serial MT\_serial.cpp

g++ -pthread -o MT\_chessboard MT\_chessboard.cpp



1. 运行程序，并将输出到相应文件

例如：在终端输入如下语句： ./MT\_serial > MT\_serial.txt

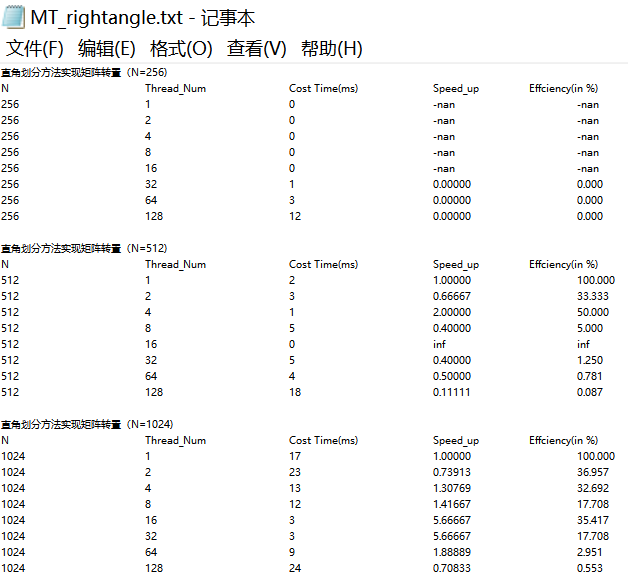
1. 实验数据分析

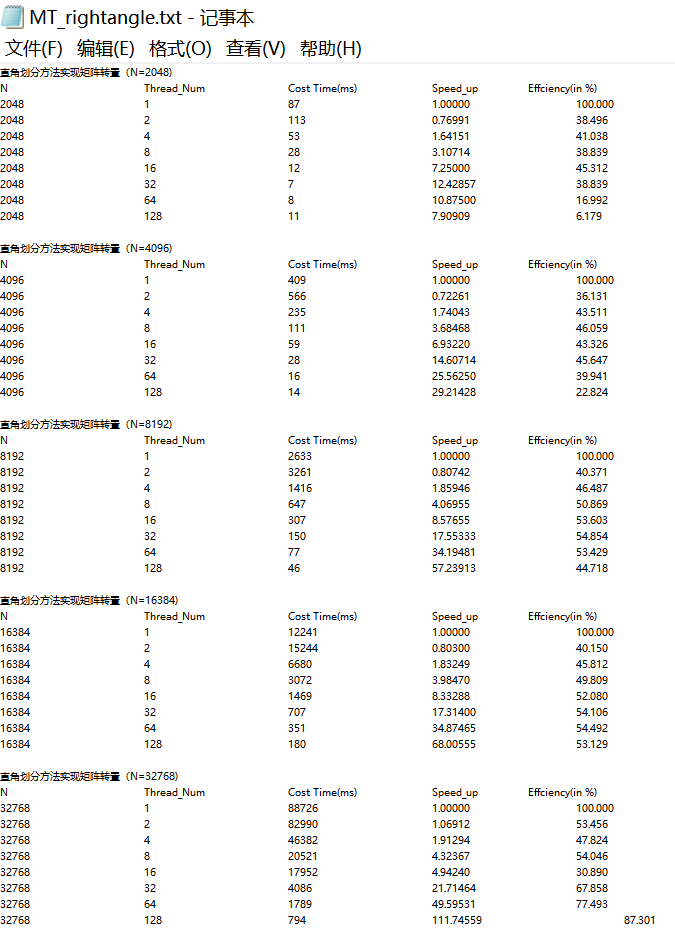
下图为串行程序的运行结果：



通过计算上面的cost time 与 N\* N的函数关系，可以得知该程序符合上述基本规律：即运算时间与计算量成正比（矩阵大小）

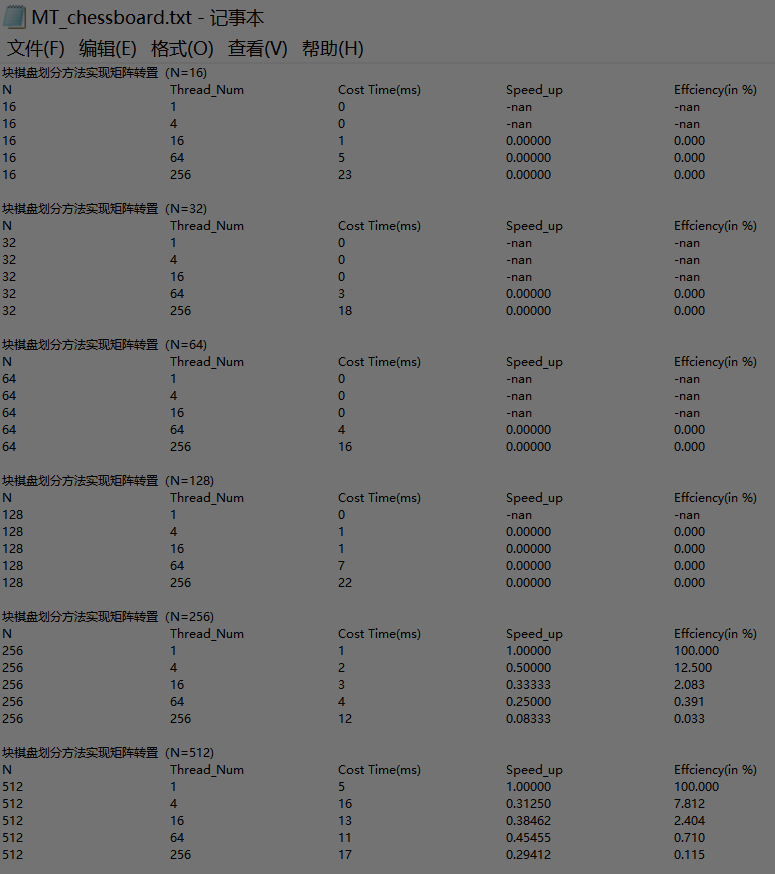
下图为使用直角划分法转置矩阵的并行程序的运行结果：

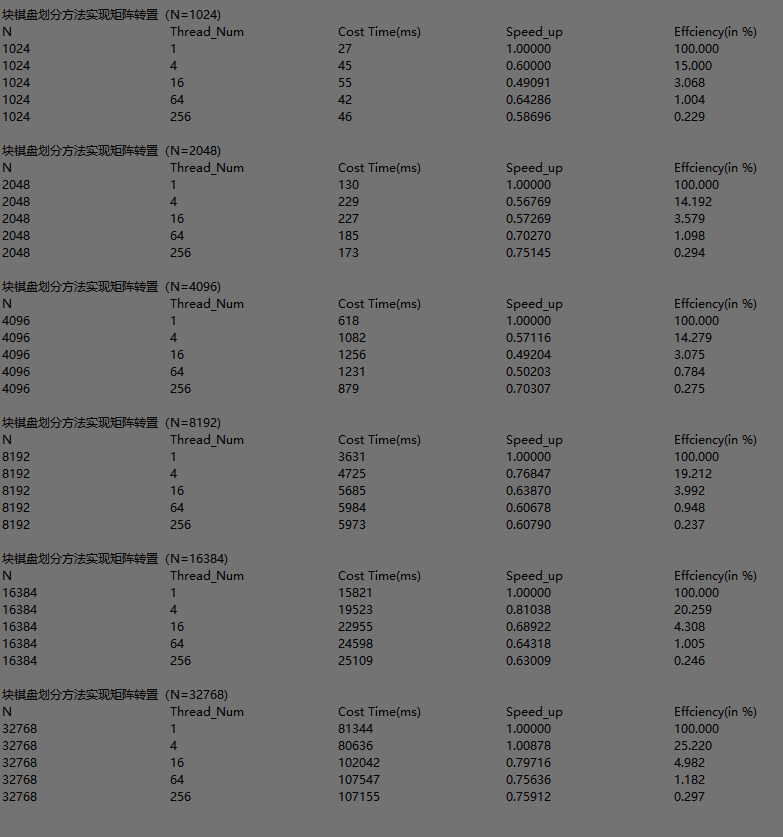




由上面两个图可知，当N与线程数的比值较小时，运算时间随线程数的增加，先减小再增加；加速比随线程数的增加，先增加再减小；效率先增加再减小。而当N与线程数的比值较大时，运算时间随线程数的增加，迅速减小

下图为使用块棋盘划分法转置矩阵的并行程序的运行结果：





由上面两个图可知，对于上述情况的某个N的特定取值，运算时间随线程数的增加而增加。

原因是：这个方法，先进行块间转置，在进行块内转置。而串行算法的计算量恰好等于用于块内转置的运算量。也就是说，这个方法的运算量是大于串行算法的，即线程数越多，划分的子块越多，多余的计算量就越多，花费的时间就会越高。

由上述分析可知，这次实验结果所表现出来的规律基本符合上次实验（多线程计算pi）得出的普遍规律：

（1）在正常范围内，计算时间随线程数的增加而减小，直至趋于平缓，于此同时，加速比也在稳步上升直至平缓。

（2）但线程是有代价的，线程是有时间消耗的（开线程，线程间通信等）。当线程数超过某个界限时，随着线程数的增加，花费的时间也越来越高，加速比也迅速下降。

（3）随着线程数的增加，效率持续递减。

（4）N越大，计算时间越长

1. 实验总结

由于课余时间有限，为了节省时间，我使用了我上次使用的的多线程程序的代码框架。并根据这次作业的要求对代码框架进行了改进优化。

首先，我迅速完成了直角划分法的实现。

但是，我很快就在块棋盘划分法遇到了困难，经过在笔记本上不断画图，理解每个变量的意义，以及相应的平面关系，最终把代码实现出来了

经过多次修改测试，我最终完成了本次实验。

经过这次实验，我意识到要想写出好的代码，就必须有扎实的基础，尤其是要打好数学基础。学习数学，虽然在短期内不会有特别明显的用途，但如果没有扎实的数学功底，遇到与数学相关的问题时，就会浪费很多无效时间在实际上不能解决该问题的尝试上。