编号

南京航空航天大学

# 毕业设计

题　目　基于 MPI 和 OpenMP 的程序性能优化研究

| | |
|---|---|
| 学生姓名 | 胡思旺 |
| 学　　号 | 161330216 |
| 学　　院 | 计算机科学与技术学院 |
| 专　　业 | 软件工程 |
| 班　　级 | 1613303 |
| 指导教师 | 陈哲　副教授 |

二〇一七年六月

# 南京航空航天大学

# 本科毕业设计（论文）诚信承诺书

本人郑重声明：所呈交的毕业设计（论文）（题目： 基于 MPI 和 openMP 的程序性能优化研究）是本人在导师的指导下独立进行研究所取得的成果。尽本人所知，除了毕业设计（论文）中特别加以标注引用的内容外，本毕业设计（论文）不包含任何其他个人或集体已经发表或撰写的成果作品。

作者签名： 年 月 日

（学号）：

# 基于 MPI 和 OpenMP 的程序性能优化研究

## 摘　要

MPI 与 openMP 编程是并行计算常用的优化方式，使用 MPI 与 openMP 编程的方式优化矩阵乘法,排序,查找最值等数据问题，并设计将矩阵乘法，排序，查找最值问题串行转化为并行的算法。最终实现 MPI,openMP 与 MPI+openMP 混合编程的三种方法实现矩阵乘法，排序，查找最值的并行优化。 MPI 与 openMP 为多处理器结点集群提供了一种有效的并行策略，openMP 采用共享内存编程模型，被用在结点内部。 MPI 采用消息传递模型，被用在集群的结点与结点之间。为研究需要，在 linux 系统下搭建一个基于两者的混合编程平台，程序在搭建的环境下能够同时实现多进程，多线程以及多进程内多线程编程。

通过对比分析矩阵乘法，排序，查找最值的并行计算处理数据效率与串行计算处理数据的效率，验证并行算法可以有效的挖掘计算机的计算能力，提高程序的计算效率。

**关键词：**MPI，openMP，MPI+openMP，并行矩阵乘法，并行排序，并行查找最值

# Research on program performance optimization based on MPI and OpenMP


## Abstract

MPI and openMP hybrid paradigms is based on parallel computing,using MPI and openMP hybrid paradigms optimised matrix multiplication, sorting, find the most value and other data problems,and design of the matrix multiplication,the sorting, to find the most value of serial algorithmto the parallel algorithm.finally achieved MPI,openMP and MPI+openMP hybrid paradigms of the method achieve matrix multiplication, sort, find the most value of parallel optimization.MPI and openMP provide a effective parallel strategy for multi processor cluster,OpenMP uses a shared memory programming model, which is used in the internal node. MPI uses the message passing model, which is used in the node and node of the cluster.For the need of the research, a hybrid programming platform based on the Linux system is built.In the process of building the environment to achieve multi process, multi-threaded and multi process multi-threaded programming.

By comparing the analysis of matrix multiplication, sorting, searching for the most value of parallel computing data processing efficiency and the efficiency of serial computing data processing,The parallel algorithm can effectively mine the computing power of the computer and improve the computational efficiency of the program


Keywords: MPI;openMP;MPI+openMP;Parallel matrix multiplication;Parallel sort;Parallel finding the most value;

# 目 录

# 第一章 引 言

## 1.1 并行计算简介

并行计算是相对于非并行计算的计算机处理数据的一种计算思想，它可以用来减少非并行计算的计算时间，提高计算机的计算效率。目前，并行计算被广泛的用于计算任务重，计算数据量大的工程领域。并行计算以成为科学家与工程师用来解决计算问题的普遍方法。目前，并行计算可以分为单机并行计算或集群多机并行计算。单机并行计算被用在支持并行计算的多处理器计算机系统中，集群多机并行计算则是由多台计算机利用网络相互连接在一起，通过不同计算机间传递计算消息来同步计算任务，最终达到并行计算效果。随着网络性能的不断提高，当前，将多台计算机利用网络相互连接构成计算集群已经成为扩展并行计算机的趋势。随着并行计算的不断发展，可跨平台的并行编程模型发展迅速。并行编程模型是一种简化并行管理而提出的高级编程语言。通过编写面向并行编程模型的程序，可以开发出跨平台的并行程序。目前应用较为广泛的并行编程模型有消息传递模型，共享变量模型以及数据并行模型。现在也已经开发出了基于上述并行编程模型的具体编程语言，其中就包括基于消息传递编程模型的 MPI 编程接口，以及基于共享变量模型的 openMP 编程接口和基于数据并行编程模型的 HPF 接口。程序员通过使用上述编程接口可以很轻松的编写出并行化程序。现如今如何找到将传统非并行化程序转变成并行化程序是当今并行计算关系的问题。许多传统的非并行化算法在通过并行化处理后，其运算的时间大大的减少，计算效率得到了质的提高。当然并不是所有的传统非并行化算法都可以找到并行化的解决方案，但随着科学技术的不断提高，相信越来越多的算法将会向并行化迈进。

## 1.2 基于 MPI 的并行编程

随着并行计算的迅速发展，可跨平台的各种并行编程模型应运而生，其中以消息传递模型尤为突出。MPI 便是以消息传递模型为架构开发出的一套跨平台，跨语言的通讯协议。MPI 是一套消息传递程序编程接口，现如今，几乎绝大多数的商业并行机都支持它。MPI 程序认

为每个参与并行计算的处理器都有属于自己的内存，并且每个处理器只能直接访问本地的指令和数据，MPI 程序利用网络将参与并行计算的处理器连接起来，不同处理器之间可以发送一个包含本地数据的消息给其他处理器，参与并行计算的处理器通过处理器之间消息的传递来同步并行计算，最终达到并行计算的目的。

运行 MPI 程序时，用户将指定参与并行的进程数，MPI 程序将进程分配到不同的处理器中运行，其中参与计算的 MPI 进程都有一个唯一的 ID 值，在 MPI 程序运行后，不同的进程可以执行不同的计算，一个进程要么执行针对其局部变量的操作，要么参与同其他进程的通讯。编写 MPI 程序的任务是如何将一个非并行化的程序通过对其计算任务的划分，将划分出来的子计算任务分配给不同 MPI 进程。不同的 MPI 进程同时执行子计算任务，最终通过消息通讯的形式来汇总任务，以此来实现提高计算效率的目的。MPI 拥有丰富的编程接口，可以满足大多数的并行计算需求，其中常用的编程接口有如下（省略函数参数与返回值）：

MPI_Init()　位于 MPI 程序最前面，用来初始化 MPI 执行环境，建立 MPI 进程之间的联系，为后续通讯做准备。

MPI_Finalize() 位于 MPI 程序最后面，用来结束 MPI 执行环境。

MPI_Comm_Rank() 用来获取每个 MPI 进程的标示 ID 值，后续通过 ID 值来划分子计算任务，是非常重要的编程接口。

MPI_Comm_Size() 用来获取相应进程组的进程数，程序根据进程数来划分计算任务。

MPI_Send() 用来向指定的 MPI 进程发送消息，用于 MPI 进程之间的通讯。

MPI_Recv() 用来接收指定 MPI 进程发送过来的消息，用于 MPI 进程之间的通讯。

MPI 程序是一种非常规格化的程序，一般形式为:

```
#include <stdio.h>
#include <mpi.h>   //包含 MPI 程序头文件
int main(int argc,char *argv[])
{
    MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank==0)
```

```
    {
        //0 号 MPI 进程分配计算任务给 1 号到 size 号 MPI 进程
    }
    else
    {
        //1 号到 size 号 MPI 进程获取计算任务并执行子计算任务
    }
    MPI_Finalize();
     return 0;
}
```

不同 MPI 进程之间通过 MPI 编程接口提供的丰富的进程通讯函数接口来实现进程之间的通讯，可见 MPI 是一个十分方便的编程接

## 1.3 基于 openMP 的并行编程

并行计算不论是在科学领域还是在工程领域都得到了迅速的发展，在各种并行计算编程模型中除了消息传递模型运用广泛了，基于共享变量模型的并行计算思想也有广泛的应用。其中 openMP 便是基于共享变量模型开发的一套并行编程框架。openMP 并不是一个简单的函数编程接口，它是一个诸多编译器支持的框架，更准确的说，它是一个并行计算编程协议，所有实现了 openMP 编程协议的编译器都可以编译 openMP 程序。基于 openMP 的并行程序不同与基于 MPI 的并行程序，openMP 是以线程的方式来实现并行计算的，不同于进程实现方式，采用线程方式来实现并行计算可以在最大程度上节约操作系统资源，同时以线程方式实现也方便不同线程之间的通讯，所以 openMP 不同与 MPI 那样拥有许多丰富通讯接口。openMP 是一个跨语言，跨平台的并行编程协议，所有实现了 openMP 的编译器都可以支持 openMP 程序，openMP 非常适用于解决简单的并行计算问题，正因如此，openMP 在并行编程领域拥有广泛的应用，将 MPI 与 openMP 结合起来编程往往有着出其不意的效果。openMP 采用一种类似编译指令的形式来指导编译器编译 openMP 并行程序，它常见的指令一下的格式:

```
 #pragma omp parallel                    //for 循环并行化代码块
{
    #pragma omp for                     //指示编译器以下 for 循环采用 openMP 并行化实现
    for (int i = 0; i < 4; ++i)
    {
        cout << i << endl;
    }
}
```

上述是一种常见的 for 循环并行化方式，但要注意的是不是所有的 for 循环都可以并行化，判断 for 循环是否可以并行化的依据是 for 循环的每一次循环之间不可以存在数据依赖，循环必须是单入口，单出口，也就是说循环内部不允许存在 break 或 goto 循环跳转语句。

为了方便并行编程 openMP 中存在许多的数据处理子句和任务调度方法，程序员可以使用不同的数据处理子句或任务调度方法来进行并行编程，openMP 的编程范式可以满足绝大多数的编程需求。

```
    for (i = 1; i < DATASIZE + 1; i++)
        {
            if (1 == (i % 2 ))
            {
    #pragma omp parallel for
                for (j = 1; j < DATASIZE + 1; j+=2)
                {
                    if (L.data[j]>L.data[j + 1])
                    {
                        int temp = L.data[j];
                        L.data[j] = L.data[j + 1];
                        L.data[j + 1] = temp;
                    }
                }
            }
            else
            {
    #pragma omp parallel for
                for (j = 2; j < DATASIZE; j += 2)
                {
```

```
                if (L.data[j]>L.data[j + 1])
                {
                    int temp = L.data[j];
                    L.data[j] = L.data[j + 1];
                    L.data[j + 1] = temp;
                }
            }
        }
    }
```

以上是采用 openMP 实现的奇偶排序的并行算法实现方式。

## 1.4 MPI 与 openMP 编程环境搭建

为了方便研究我采用在 linux 系统上使用 c/c++语言来进行并行编程开发，linux 系统本身不带有 MPI 与 openMP 编程库，我需要为 linux 配置 MPI 与 openMP 编程环境。

第一步:安装 g++编译器

在 debian 下控制台下输入 sudo apt-get install g++命令根据提示可以安装好 g++编译器，g++编译器最好安装最新版，一些老版本的 g++编译器不支持 openMP 编译指令，无法编译 openMP 程序。

第二步:安装 MPI 编程环境

到 MPI 官网 http://www.mpich.org/static/downloads/下载对应版本的 MPI 安装包，安装包下载完成后使用 tar 命令将安装包解压，将解压下来的包放在自己固定的目录下面，这样可以在控制台下直接输入 MPI 命令。进入解压包执行 make&make install 命令，这样便安装好了 MPI 软件。 安装后加入环境变量/etc/profile，并执行 source /etc/profile。

在/etc/profile 文件下加入一下路径，注意是自己 MPI 的安装路径。

PATH=$PATH:(MPI 安装路径)

MANPATH=$MANPATH:(MPI 安装路径)

export PATH MANPATH

在控制台下输入 mpirun,mpic++命令,如果命令找到了说明 MPI 开发环境搭建成功。

至此，便完成了 MPI 与 openMP 环境的搭建。接下来我们便可以进行程序开发工作了。

# 第二章 并行矩阵乘法

## 2.1 非并行矩阵乘法算法

矩阵乘法运算是线性代数中常见的矩阵运算，它是其它线性代数运算的基础，它只有在第一个矩阵的列数和第二个矩阵的行数相同时才有计算的意义，否则两个矩阵不能进行相乘运算。

设矩阵 M 为 m×p 的矩阵，矩阵 N 为 p×n 的矩阵，那么矩阵 M 与矩阵 N 相乘将得到 m*n 的矩阵 R。记为 R=MN，其中矩阵 R 中的第 i 行第 j 列可以用下列公式表示:

$$(MN)_{ij}=\sum_{k=1}^{p} a_{ik}b_{kj}=a_{i1}b_{1j}+a_{i2}b_{2j}+\ldots+a_{ip}b_{pj}$$

### 2.1.1 非并行矩阵乘法程序实现

```
void matrixMulti(Matrix_Result &result,Matrix_M &M,Matrix_N &N)
{
    if (HIGH_M != WIDTH_N)
        return;
    else
    {
        int i, j, k;
        for (i = 0; i < WIDTH_M; i++)
        {
            for (j = 0; j < HIGH_N; j++)
            {
                int sum = 0;
                for (k = 0; k < WIDTH_N; k++)
```

```
                    {
                        sum += (M.data[i][k]*N.data[k][j]);
                    }
                    result.data[i][j] = sum;
                }
            }
        }
    }
```

### 2.1.2 程序分析

为了方便计算，我们定义三个自定义类型 Matrix_Result,Matrix_M,Matrix_N,这三个自定义的数据类型中都包含有一个用于存储矩阵数据的二维数组，Matrix_M 与 Matrix_N 分别存储两个满足矩阵乘法规则的待计算矩阵，Matrix_Result 用来存储矩阵相乘后得到的矩阵。本章节中所有涉及的矩阵乘法运算都涉及这三个自定义的数据类型，往后不在解释。

下面我们来对整个程序进行分析，在程序中我们定义了两个全局变量 HIGH_M ，WIDTH_N 分别代表第一个矩阵的列数和第二个矩阵的行数。

```
if (HIGH_M != WIDTH_N)
        return;
```

程序先判断两个矩阵是否满足计算要求，如果不满足计算要求，程序便返回，不做矩阵乘法计算。通过三层迭代循环计算两个矩阵相乘。我们通过以第一个矩阵 M 行的遍历，将矩阵 M 的第 i 行与矩阵 N 的第 j 列数据逐个相乘并计算总和便得到矩阵 Result 第 i 行第 j 列的数据。

在矩阵乘法计算中我们参考如下的计算公式：

$$(MN)_{ij}=\sum_{k=1}^{p} a_{ik}b_{kj}=a_{i1}b_{1j}+a_{i2}b_{2j}+\ldots+a_{ip}b_{pj}$$

对于非并行的矩阵乘法可以用 c 语言非常简单的实现，算法简单易懂。矩阵 M 与矩阵 N 做相乘运算得到矩阵 R,程序通过遍历矩阵 M 的行与矩阵 N 的列，将其相乘后得到对应矩阵 R

的矩阵值。由于矩阵相乘不存在数据依赖问题，所以矩阵乘法可以使用并行算法进行优化，以此来提高矩阵相乘的运算效率。

## 2.2 基于 MPI 的矩阵乘法优化

MPI 是以消息传递模型为架构开发出的一套跨平台，跨语言的通讯协议。MPI 编程接口在并行计算中使用非常广泛。可以使用 MPI 编写出并行矩阵优化算法。对于矩阵 M 与矩阵 N，在满足矩阵乘法运算的前提下，我们知道矩阵乘法的规则为矩阵 M 的行各个值相应乘以矩阵 N 相应列对应的值。运算规则为:

设矩阵 M 为:

$$M=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & & & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

设矩阵 N 为:

$$N=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & & & b_{2n} \\ \vdots & & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

矩阵 R=(MN),运算规则为:

$$(MN)_{ij}=\sum_{k=1}^{p} a_{ik}b_{kj}=a_{i1}b_{1j}+a_{i2}b_{2j}+\ldots+a_{ip}b_{pj}$$

为了实现并行计算的要求需要对相乘的矩阵进行数据的划分，我们定义一种这样的划分规则。以矩阵 M 的行数进行子计算任务的划分，假设矩阵 M 有 x 行，MPI 计算进程数为 n，那么每个进程需要计算的行数为 x/n 行，将这些函行数分配给 MPI 进程，让它们各个执行自己的计算任务。

假设 M 矩阵与 N 矩阵为:

$$M=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} N=\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

假设我们存在两个 MPI 进程，则第一个进程负责计算:

$$R_1=MN=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix}*\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

第二个 MPI 进程负责计算:

$$R_2=MN=\begin{bmatrix} a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}*\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

由于采用这样的划分方法不同进程之间不存在数据依赖。最终我们将 R1 与 R2 矩阵按照矩阵计算的顺序要求将数据合并即可。

### 2.2.1 MPI 实现矩阵乘法程序实现

```
void matrixMultiParallel(int argc,char * argv[],Matrix_Result &result, Matrix_M &M, Matrix_N &N)
```

```
{
    int rank,size;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size==1)
    {
        matrixMulti(result, M, N);      //如果是一个 MPI 进程，则直接使用非并行算法计算
    }
    else
    {
        if(rank==0)
        {
            int dest,source,row,offset,flag;
            row = WIDTH_M / size;
            offset=0;
            source=0;
            for (dest = 1; dest < size; dest++)
            {
                MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
                MPI_Send(&row, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
                offset = offset + row;
            }
            for (int i = offset; i < WIDTH_M; i++)
            {
                for (int j = 0; j < HIGH_N; j++)
                {
```

```
                int sum = 0;
                for (int k = 0; k < WIDTH_N; k++)
                {
                    sum += (M.data[i][k]*N.data[k][j]);
                }
                result.data[i][j] = sum;
            }
        }
        for(dest = 1;dest < size; dest++)
        {
            MPI_Recv(&flag, 1, MPI_INT, dest, 2, MPI_COMM_WORLD, &status);
        }
    }
    if(rank>0)
    {
        int dest,source,row,offset;
        source=0;
        MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&row, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        for (int i = offset; i < (offset+row); i++)
        {
            for (int j = 0; j < HIGH_N; j++)
            {
                int sum = 0;
                for (int k = 0; k < WIDTH_N; k++)
                {
                    sum += (M.data[i][k]*N.data[k][j]);
                }
```

```
                    result.data[i][j] = sum;
                }
            }
            MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD);
        }
        MPI_Finalize();
    }
}
```

### 2.2.2 程序分析

程序运行必须包含预处理指令，也就是说必须包含有 MPI 标准头文件 mpi.h 和 c 语言标准输入输出文件 stdio.h。MPI 程序运行时必须知道 argc 和 argv 参数，因此我们将这两个参数以函数参数形式传入我们自己写的 matrixMultiParallel 函数中。

```
MPI_Init(&argc, &argv);
```

首先，我们先进行 MPI 程序的初始化

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

其次我们通过 MPI 的标准编程接口获取当前 MPI 进程标识号与 MPI 进程总数并存入我们自定义的变量 rank 与 size 中，以作为后续任务划分的参考依据。

接下来程序判断 size 是否为 1，如果是 1 个进程，则我们直接调用非并行的矩阵乘法程序，如果 size 不为 1，也就是说采取并方式。接下来便是十分重要的环节，我们需要对任务进行划分，将计算任务分配到各个不同进程中计算。划分工作在 0 号 MPI 进程中进行。划分根据进程号进行。

```
for (dest = 1; dest < size; dest++)
{
    MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
    MPI_Send(&row, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
    offset = offset + row;
}
```

通过 MPI 提供的 MPI_Send 函数将划分单位发送给各个 MPI 进程，同时主进程也需要计算一部分任务。我们采取以矩阵 M 的行划分，也就是说将每个进程需要计算的行数 row,与当前行数与矩阵 M 的第一行的偏离 offset 发送给各个 MPI 进程。

在 rank!=0 的 MPI 进程中

```
MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
MPI_Recv(&row, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
for (int i = offset; i < (offset+row); i++)
{
    for (int j = 0; j < HIGH_N; j++)
    {
        int sum = 0;
        for (int k = 0; k < WIDTH_N; k++)
        {
            sum += (M.data[i][k]*N.data[k][j]);
        }
        result.data[i][j] = sum;
    }
}
MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD);
```

我们通过 MPI 提供的 MPI_Recv 函数接受来自 rank 为 0 的 MPI 进程发送过来的 row 与 offset，并计算当前 MPI 进程分配的计算任务，计算结束后发送一个信号给 0 号 MPI 进程 MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD)，告诉它计算完毕，以便 0 号进程统计计算时间。当 0 号进程收集到所有 MPI 进程发送过来的计算结束消息后便知道任务全部计算成功了。

```
for(dest = 1;dest < size; dest++)
    MPI_Recv(&flag, 1, MPI_INT, dest, 2, MPI_COMM_WORLD, &status);
```

有关 MPI 函数使用说明，请见参考附录。

## 2.3 基于 openMP 的矩阵乘法优化

openMP 采用线程的方式来实现并行计算，由于线程之间可以实现资源的共享，它比起 MPI 来更加轻量级。openMP 对于简单的并行计算问题提提供了可行的解决方案。

对于矩阵 M 和矩阵 N:

$$M=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} N=\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

我们采用类似与 MPI 实现的方式一样对两个矩阵进行数据划分。不过用于 openMP 不同与 MPI 无法实现指定的划分，它是通过简单的编译指令，起子计算的划分交由编译器完成，我们只要给定划分规则即可，无需关系那个线程计算那个模块的问题。向比于 MPI，使用 openMP 实现的方案，代码看起来更加的简单易懂。

### 2.3.1 openMP 实现并行矩阵乘法程序实现

```
void matrixMultiOmpParallel(int argc,char * argv[],Matrix_Result &result, Matrix_M &M,
Matrix_N &N)
{
    if(argc==2)
    {
        printf("请输入线程数!\n");
    }
    else
    {
        int thread=atol(argv[2]),i,j,k;
        omp_set_num_threads(thread);
```

```
        #pragma omp parallel shared(result,M,N) private(i,j,k)
        {
            #pragma omp for schedule(dynamic)
            for(i=0;i<DATASIZE;i++)
            {
                for(j=0;j<DATASIZE;j++)
                {
                    result.data[i][j]=0;
                    for(k=0;k<DATASIZE;k++)
                    {
                        result.data[i][j]+=M.data[i][k]*N.data[k][j];
                    }
                }
            }
        }
    }
}
```

### 2.3.2 程序分析

对于 openMP 实现的并行矩阵乘法，由于 openMP 简单易用，我们可以发现它与非并行程序差异不大，在程序运行前我们使用 openMP 标准函数 omp_set_num_threads(thread);以此来设置 openMP 运行的线程数。

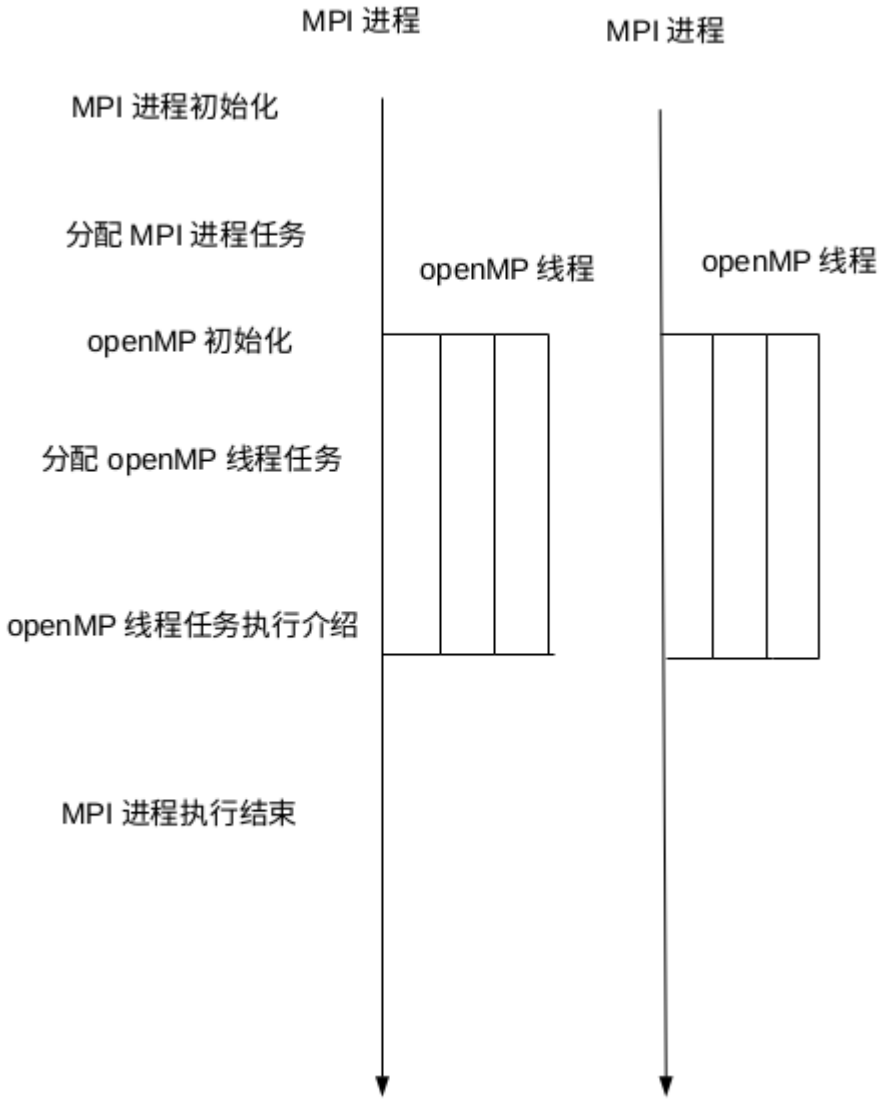
在三层迭代的 for 循环中我们在最外层循环添加 openMP 编译指令#pragma omp parallel shared(result,M,N) private(i,j,k)，这样便通知支持 openMP 编译的编译器，该层 for 循环可以进行 openMP 线程并行优化，也就是说该层 for 循环的每次迭代是数据无关的可以交给不同的线程去并行执行。

```
    #pragma omp for schedule(dynamic)
     for(i=0;i<DATASIZE;i++)
```

## 2.4 基于 MPI 与 openMP 混合编程的矩阵乘法优化

使用 MPI 或使用 openMP 进行矩阵乘法的并行优化都可以达到预期的优化效果，但是我们可以采用 MPI 与 openMP 混合编程的技术来进行最大的优化，首先使用 MPI 在进程的层面上进行数据的划分，在各个 MPI 进程中再次使用 openMP 对 MPI 分配的计算任务再次进行 openMP 层面上的划分，如此便可以对算法再次优化，可以达到最大的计算效率。

MPI 与 openMP 并行编程模型如下(假设只有两个 MPI 进程):




采用 MPI 与 openMP 混合编程可以同时使用两者的优点，达到并行计算的最大优化效果。使用 MPI 与 openMP 混合编程在计算效率上比单纯使用 MPI 编程有所提高。

2. 4. 1 混合编程实现矩阵乘法程序实现

```
void matrixMultiParallel(int argc,char * argv[],Matrix_Result &result, Matrix_M &M,
Matrix_N &N)
{
    int rank,size;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size==1)
    {
        matrixMulti(result, M, N);
    }
    else
    {
        if(rank==0)
        {
            int dest,source,row,offset,flag;
            row = WIDTH_M / size;
            offset=0;
            source=0;
            for (dest = 1; dest < size; dest++)
            {
                MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
                MPI_Send(&row, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
                offset = offset + row;
            }
            #pragma omp parallel shared(result,M,N)
             {
```

```
            #pragma omp for schedule(dynamic)
            for (int i = offset; i < WIDTH_M; i++)
            {
                for (int j = 0; j < HIGH_N; j++)
                {
                    int sum = 0;
                    for (int k = 0; k < WIDTH_N; k++)
                    {
                        sum += (M.data[i][k]*N.data[k][j]);
                    }
                    result.data[i][j] = sum;
                }
            }
        }
    for(dest = 1;dest < size; dest++)
    {
        MPI_Recv(&flag, 1, MPI_INT, dest, 2, MPI_COMM_WORLD, &status);
    }
}
if(rank>0)
{
    int dest,source,row,offset;
    source=0;
    MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&row, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
     #pragma omp parallel shared(result,M,N)
     {
        #pragma omp for schedule(dynamic)
```

```
                for (int i = offset; i < (offset+row); i++)
                {
                    for (int j = 0; j < HIGH_N; j++)
                    {
                        int sum = 0;
                        for (int k = 0; k < WIDTH_N; k++)
                        {
                            sum += (M.data[i][k]*N.data[k][j]);
                        }
                        result.data[i][j] = sum;
                    }
                }
            }
            MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD);
        }
        MPI_Finalize();
    }
}
```

### 2.4.2 程序分析

程序运行前与先前一样必须包含预处理指令，也就是说必须包含有 MPI 标准头文件 mpi.h 和 c 语言标准输入输出文件 stdio.h。MPI 程序运行时必须知道 argc 和 argv 参数，因此我们将这两个参数以函数参数形式传入我们自己写的 matrixMultiParallel 函数中。

MPI_Init(&argc, &argv);

首先，我们先进行 MPI 程序的初始化

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &size);

其次我们通过 MPI 的标准编程接口获取当前 MPI 进程标识号与 MPI 进程总数并存入我们自定义的变量 rank 与 size 中，以作为后续任务划分的参考依据。

在任务划分环节中我们采用的方法与基于 MPI 的矩阵乘法优化相同，算法可以参考本论文的 2.2.2 节。

由于我们关注的重点是如何进行 MPI 与 openMP 的混合编程，两者的混合编程模型我们在前面章节中以有所解释，我们只要在各个 MPI 进程的计算任务中加入 openMP 编译指令，这样可以告诉编译器采用多线程的方式运行该计算任务。

```
MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
MPI_Recv(&row, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
#pragma omp parallel shared(result,M,N)
{
    #pragma omp for schedule(dynamic)
    for (int i = offset; i < (offset+row); i++)
    {
        for (int j = 0; j < HIGH_N; j++)
        {
            int sum = 0;
            for (int k = 0; k < WIDTH_N; k++)
            {
                sum += (M.data[i][k]*N.data[k][j]);
            }
            result.data[i][j] = sum;
        }
    }
}
MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD);
```

我们在 rank 不等于 0 的 MPI 进程中获取各个计算任务，然后在执行计算任务时采取 openMP 的方式执行即可，这样便实现了 MPI 与 openMP 的混合编程实现矩阵乘法。

## 2.5 并行优化对比分析

到目前为止我们采用 MPI,openMP,MPI 与 openMP 混合编程实现了矩阵乘法的并行优化，现在为验证算法有效性，我们分别采用 500*500,1000*1000,1500*1500 的矩阵来分别验证 MPI,openMP,MPI 与 openMP 混合编程实现的程序效果。

实验环境为 ubuntu12.04 系统,linux 内核版本号为 Linux version 3.8.0-38-generic，处理器为 intel Xeon(R) E5620@2.4GHz*16。

下面的表格分别列出了 MPI，openMP，MPI 与 openMP 混合编程实现的并行矩阵乘法在实验机器上的时间统计。

### 2.5.1 数据统计

下列表格统计了基于 MPI 的矩阵乘法优化的程序运行时间统计

| MPI 进程数 | 500*500 | 1000*1000 | 1500*1500 |
|---|---|---|---|
| 1 | 1.093443(s) | 10.184872(s) | 40.59069(s) |
| 2 | 0.543956(s) | 5.121438(s) | 21.273153(s) |
| 4 | 0.273116(s) | 3.250605(s) | 10.439778(s) |
| 8 | 0.262404(s) | 2.792285(s) | 10.176009(s) |
| 12 | 0.202713(s) | 2.058928(s) | 7.328876(s) |

表 2.1 MPI 实现并行矩阵乘法时间统计表格

下列表格统计了基于 openMP 的矩阵乘法优化的程序运行时间统计

| openMP 线程数 | 500*500 | 1000*1000 | 1500*1500 |
|---|---|---|---|
| 1 | 1.62582(s) | 14.817589(s) | 63.655734(s) |
| 2 | 0.818937(s) | 7.233324(s) | 30.294531(s) |
| 4 | 0.415416(s) | 3.619727(s) | 14.816831(s) |
| 8 | 0.23732(s) | 2.346853(s) | 9.20865(s) |
| 12 | 0.221558(s) | 1.934575(s) | 5.12986(s) |

表 2.2 openMP 实现并行矩阵乘法时间统计表格

下列表格统计了基于 MPI 与 openMP 混合编程的矩阵乘法优化的程序运行时间统计

| MPI 进程数为 1 | | | |
|---|---|---|---|
| openMP 线程数 | 500*500 | 1000*1000 | 1500*1500 |
| 1 | 1.096371(s) | 10.150751(s) | 39.589899(s) |
| 2 | 0.543623(s) | 5.082305(s) | 19.86103(s) |
| 4 | 0.277553(s) | 2.541233(s) | 9.600367(s) |
| 8 | 0.181814(s) | 1.695404(s) | 6.062196(s) |

表 2.3.1 MPI 与 openMP 混合编程实现并行矩阵乘法优化实验数据图（进程为 1）

| MPI 进程数为 2 | | | |
|---|---|---|---|
| openMP 线程数 | 500*500 | 1000*1000 | 1500*1500 |
| 1 | 0.545032(s) | 5.169813(s) | 20.373336(s) |
| 2 | 0.278146(s) | 2.593431(s) | 10.302757(s) |
| 4 | 0.165692(s) | 1.668729(s) | 10.281692(s) |
| 8 | 0.158914(s) | 1.436804(s) | 5.520249(s) |

表 2.3.2 MPI 与 openMP 混合编程实现并行矩阵乘法优化实验数据图（进程为 2）

| MPI 进程数为 4 | | | |
|---|---|---|---|
| openMP 线程数 | 500*500 | 1000*1000 | 1500*1500 |
| 1 | 0.277617 | 3.623034 | 10.154049 |
| 2 | 0.184081 | 2.621678 | 7.612473 |
| 4 | 0.155751 | 1.435125 | 5.92285 |

表 2.3.3 MPI 与 openMP 混合编程实现并行矩阵乘法优化实验数据图（进程为 4）

通过对实验数据的分析比较,我们发现随着 MPI 进程数或 openMP 线程数的增加，程序的运行时间大幅度的减少，由此可以证明并行算法确实比非并行算法在程序运行效率上有着明显的优势。

为方便比对数据，我们通过对实验数据作图分析来对比实验数据。

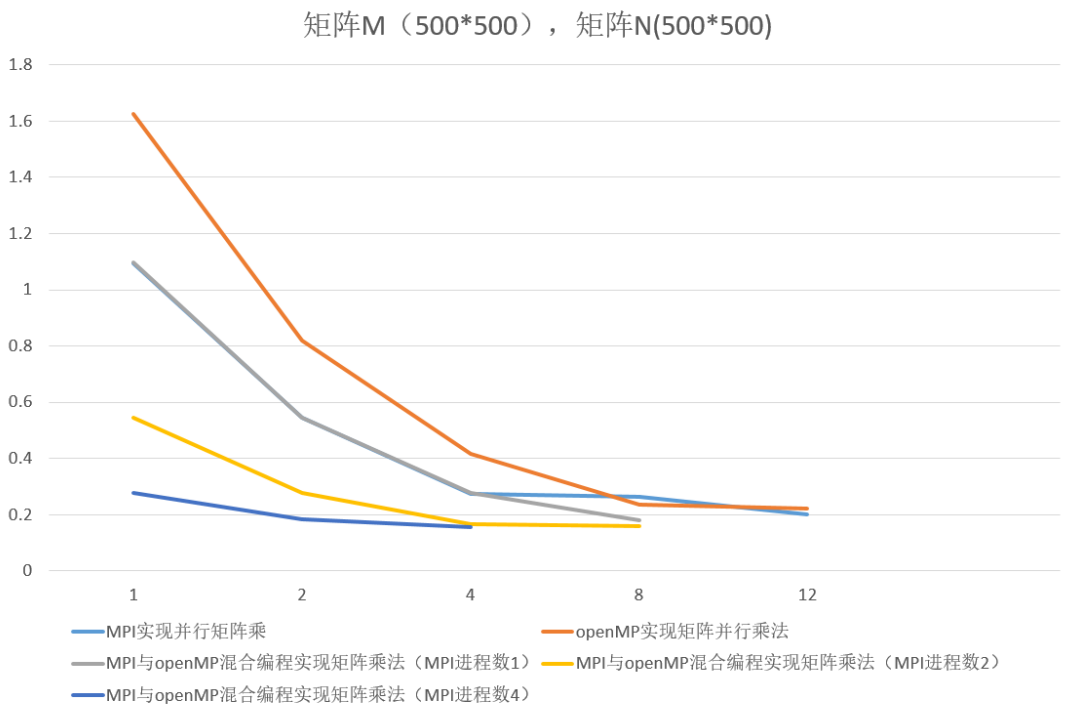



图 2.1 矩阵大小为 500*500 的实验对比图

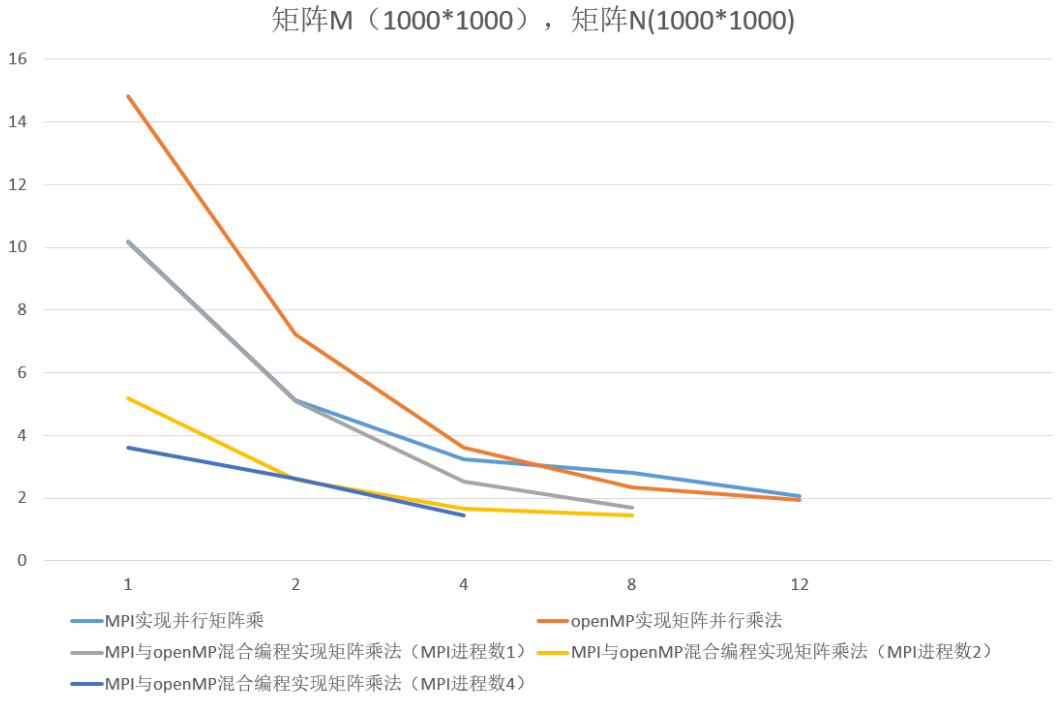



图 2.2 矩阵大小为 1000*1000 的实验对比图

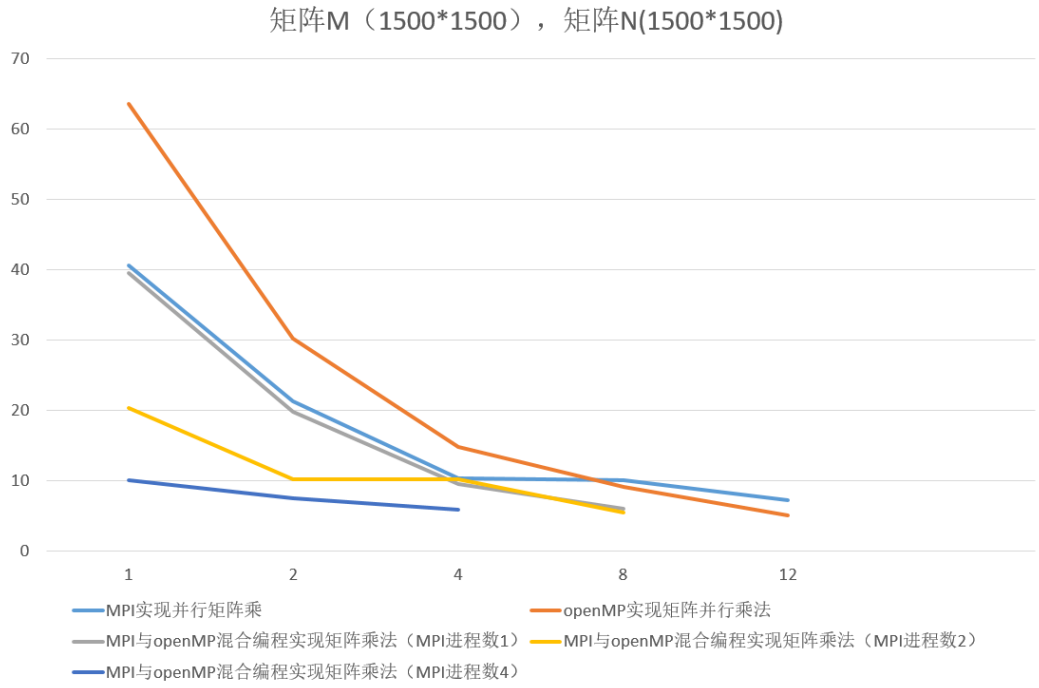



图 2.3 矩阵大小为 1500*1500 的实验对比图

### 2.5.2 理论分析

首先，我们通过对比单独使用 MPI 与单独使用 openMP 优化程序来分析实验数据。

通过对实验数据的对比分析，我们发现在单独使用 MPI 优化矩阵乘法时，随着 MPI 进程数的增加，也就是说在并行程度的不断提高下，程序的运行时间有着显著的减少。在单独对比 openMP 实验数据时也可以得出一样的结论，我们发现在实验 openMP 时，由于线程比进程在操作系统资源消耗上比较少，随着 openMP 线程数的不断增加，采用 openMP 比单独采用 MPI 在程序运行时间上有着显著的区别,尽管在并行程序小于 4 是采用 MPI 在时间上比采用 openMP 要少，但随着将并行程度提高到 8 个以上时，openMP 的优势便显示出来了。因为单独使用 openMP 优化程序时，当只要求 2 个线程时，操作系统需要维护一个进程，两个线程。单独使用 MPI 优化程序时，但要求 2 个进程时，操作系统需要维护两个进程，故此，并行程度不高时，采用 MPI 比采用 openMP 好。但是随着并行程度的提高，比如要求为 8，也就是说单独使用 openMP 优化程序时，当只要求 8 个线程时，操作系统需要维护一个进程，8 个线程，单独使用 MPI 优化程序时，在要求 8 个进程时，操作系统需要维护 8 个进程，因为操作系统对进程的维护比对线程的维护消耗的资源更多，造成程序运行时间的增加。

现在我们得出了一个重要结论，在并行任务分配不多的情况下，也就是说进程数或线程数不多时，使用 MPI 比使用 openMP 程序运行效果更好。

其次，我们通过对比使用 MPI 与 openMP 混合编程与单独使用 MPI 或单独使用 openMP 来分析实验数据。

通过对实验数据的对比我们发现在进程数相同的情况下使用 MPI 与 openMP 混合编程比单独使用 MPI 编程在程序运行效率上有所不同，使用混合编程时，随着 openMP 线程的增加，使用 MPI 与 openMP 混合编程比单独使用 MPI 编程在时间上更少，使用混合编程时随着 MPI 进程数与 openMP 线程数的增加，程序的运行时间有着明显的减少，程序的运行效率在飞速的提升。

至此，我们可以得出结论在使用 MPI 与 openMP 混合编程时，程序的运行效率比单独使用 MPI 或单独使用 openMP 更好，我们在矩阵乘法的并行优化时应优先采用 MPI 与 openMP 混合编程模式。

## 2.6 总结

通过以上的数据分析我们得出了两个结论：

第一， 在对矩阵乘法并行优化时，在并行任务分配不多的情况下，也就是说进程数或线程数不多时，使用 MPI 比使用 openMP 程序运行效果更好。在并行任务分配多的情况下，也就是说进程数或线程数多时，使用 openMP 比使用 MPI 程序运行效果更好。

第二， 使用 MPI 与 openMP 混合编程时，程序的运行效率比单独使用 MPI 或单独使用 openMP 更好，我们在矩阵乘法的并行优化时应优先采用 MPI 与 openMP 混合编程模式。

# 第三章 并行求最值

## 3.1 非并行求最值算法

在计算机中求序列的最值是一种非常常见的操作，它是其它复杂算法的基础。我们定义一个序列:

$$L=\{a_1,a_2,a_3,\cdots\cdots\cdots a_n\}$$

序列 L 中的数据是同一类型的数据，它们有序或无序，对于非并行求序列最值的算法，我们假设最值为 a1，然后通过遍历序列，通过比较序列中其他数据与 a1 的大小，最终确定序列的最值。

### 3.1.1 非并行求最值程序实现

```
int Max(List &L)
{
    int max = L.data[0];
    for (int i = 0; i < DATASIZE; i++)
    {
        if (L.data[i] > max)
        {
            max = L.data[i];
        }
    }
    return max;
}
```

通过对算法的分析，可以得出算法时间复杂度为 O(N),空间复杂度为 O(1)。

### 3.1.2 程序分析

为了方便计算，我们定义一个自定义类型 List，List 中包含一个数组 data 用来存储待求最值得序列数据，在非并行求序列最值的程序中：

```
int max = L.data[0];
```

我们先假定序列最值为第一个元素，通过对序列的遍历：

```
for (int i = 0; i < DATASIZE; i++)
{
    if (L.data[i] > max)
    {
        max = L.data[i];
    }
}
```

不断将序列中的元素与预先假定的最值对比，最终求出序列的最值。可以看出非并行求序列最值得程序简单易懂，下面我们将使用 MPI 与 openMP 来实现求序列最值得并行化处理。

## 3.2 基于 MPI 的求最值优化

MPI 做为一种基于消息通讯模型的并行编程接口，我们可以使用 MPI 来实现对序列求最值进行并行化处理，以此可以提高计算效率，极大的缩短求序列最值的时间。

假设对于任意给定的序列 L:

$$L=\{a_1,a_2,a_3,\cdots\cdots\cdots a_n\}$$

序列 L 中的数据量为 N,假设参与并行计算的 MPI 进程数为 n,则我们可以对序列 L 进行划分，划分的依据为进程数 n，这样每个进程处理的序列数据量缩减为 N/n。也就是说，每个 MPI 进程求数据量 N/n 的序列最值，然后将所有 MPI 进程求出的最值交给主 MPI 进程，由主 MPI 进程求出所有其他 MPI 进程最值中的最值，这样便求出整个序列 L 的最值。以此，我们便实现了求序列最值的并行化处理。

举一个实例，假设我们有两个 MPI 进程，则将序列 L 划分为:

$$L_1=\{a_1,a_2,a_3,\cdots\cdots\cdots a_{n/2}\}\ L_2=\{a_{n/2+1},a_{n/2+2},a_{n/2+3},\cdots\cdots\cdots a_n\}$$

第一个 MPI 进程处理序列 L1，求出序列 L1 的最值 M1。第二个 MPI 进程处理序列 L2，求出序列 L2 的最值 M2。然后第二个 MPI 进程将 M2 发送给第一个进程，第一个比较 M1 与 M2 的大小最终求出整个序列 L 的最值。

### 3.2.1 MPI 实现求序列最值程序实现

```
void MaxParallel(int argc,char * argv[],List &L, keyType &max)
{
    max = L.data[0];
    MPI_Status status;
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size == 1)
    {
        max = Max(L);
    }
    else
    {
        if (rank == 0)
        {
            int dest, start, end, length, source, i,temp;
            start = 0;
            length = DATASIZE / size;
            end = start + length;
            for (dest = 1; dest < size; dest++)
            {
                MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
```

```
            if(start+length<DATASIZE)
                MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            else
                MPI_Send(&L.length, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            start += length;
            end += length;
        }
        for (i = start; i < end; i++)
        {
            if (L.data[i] > max)
                max = L.data[i];
        }
        for (i = 1; i < size; i++)
        {
            source = i;
            MPI_Recv(&temp, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
            if (max < temp)
                max = temp;
        }
    }
    if (rank > 0)
    {
        int start, end, length, dest,source, i, temp;
        source = 0;
        dest = 0;
        MPI_Recv(&start, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&end, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        for (i = start; i < end; i++)
```

```
            {
                temp = L.data[i];
                if (L.data[i] > temp)
                    temp = L.data[i];
            }
            MPI_Send(&temp, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
        }
        MPI_Finalize();
    }
}
```

### 3. 2. 2 程序分析

基于 MPI 的求最值程序实现方式我们从上一节中可以看到，我们通过实现一个 MaxParallel 函数来完成计算任务，该函数有 4 个参数：

argc,argv 参数通过 main 函数传入，以用来初始化 MPI 程序。

List 参数包含有待求最值得序列数据。

Max 参数返回求得的最值。

为了运行该程序，我们必须包含 MPI 标准头文件 mpi.h 和 c 语言输入输出标准头文件 stdio.h。程序开始是以

```
MPI_Init(&argc, &argv);
```

该函数是 MPI 标准接口函数，用来初始化 MPI 运行环境。

接下来我们通过 MPI 的另外两个函数来获取 MPI 进程标识符和 MPI 进程总数。

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

接下来便是程序最重要的一部分，我们首先判断 MPI 进程总数，如果 MPI 进程总数为 1，说明我们不需要使用并行化操作，则直接调用非并行化程序，如果 MPI 进程总数不为 1，则采取并行化方式运行程序。

```
if (size == 1)
{
```

```
    max = Max(L);
}
```

接下来我们开始根据 MPI 进程标识符来操作数据，首先如果是在 rank=0 进程中，我们先要求序列 L 进行划分，将其划分为不同的子序列，并将一些必要的划分依据发送给其他 MPI 进程。划分依据为 start,end 下标，也就是说主进程将划分的下标 start,end 两个子序列界限位发送给其他 MPI 进程。

```
MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
if(start+length<DATASIZE)
    MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
else
    MPI_Send(&L.length, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
```

这样其他 MPI 进程负责处理下标从 start 到 end 位置的最值，每个 MPI 进程求出下标从 start 到 end 位置的最值后将最值发送给主 MPI 进程，主 MPI 进程也负责计算一部分子序列的最值，主进程负责接收其他 MPI 进程发送过来的最值，并求出所有子序列最值中的最值，这样便求出了整个序列 L 的最值。

0 号 MPI 进程接收其他进程求出的字序列最值，并在求出所有子序列最值得最值。

```
for (i = 1; i < size; i++)
{
    source = i;
    MPI_Recv(&temp, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
    if (max < temp)
        max = temp;
}
```

## 3.3 基于 openMP 的求最值优化

openMP 采用线程方式来实现并行计算需求，openMP 是一种简便的并行计算解决方案，我们虽然无法指定线程处理的数据位置，但这确对并行计算的需求并没有影响。

对于任意给定的序列 L:

$$L=\{a_1,a_2,a_{3,}\cdots\cdots\cdots a_n\}$$

3.3.1 openMP 实现求序列最值程序实现

```
void OmpMaxParallel(List &L,keyType &max)
{
    max = L.data[0];
    #pragma omp parallel for
    for (int i = 0; i < DATASIZE; i++)
    {
        if (L.data[i] > max)
        {
            max = L.data[i];
        }
    }
}
```

3.3.2 程序分析

openMP 采用编译指令的方式来进行并行优化，它的操作十分的简单，与非并行化程序在程序结构上十分相似，为了实现 openMP 并行化，我们只需要在可以并行化的 for 循环代码块上加入 openMP 编译指令#pragma omp parallel for

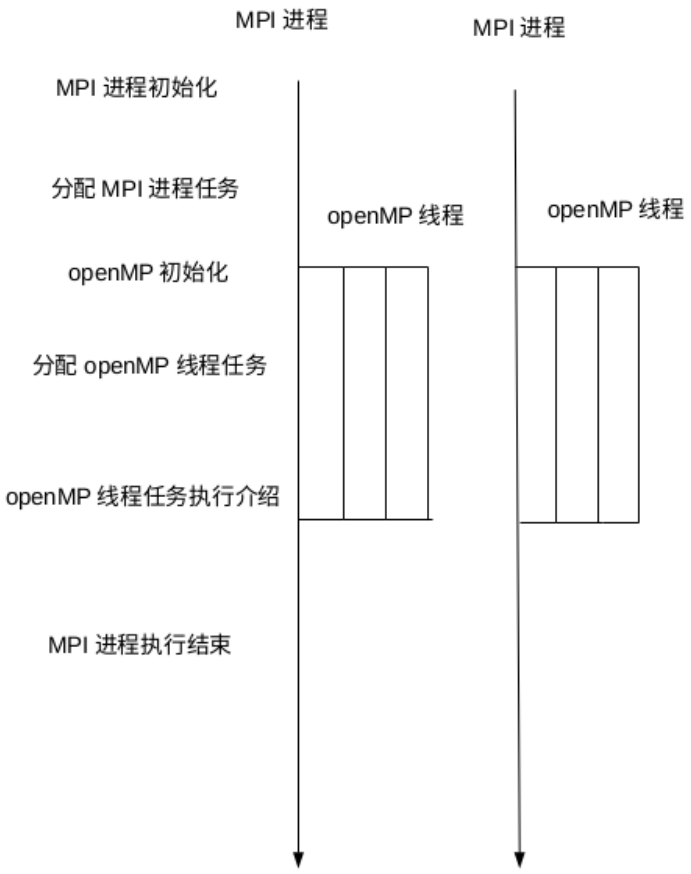
```
#pragma omp parallel for
    for (int i = 0; i < DATASIZE; i++)
    {
        if (L.data[i] > max)
        {
            max = L.data[i];
        }
    }
```

## 3.4 基于 MPI 与 openMP 混合编程的求最值优化

对于给定任意序列 L，我们已经实现了 MPI 与 openMP 的并行化优化

$$L=\{a_1,a_2,a_3,\cdots\cdots\cdots a_n\}$$

对于 MPI 与 openMP 混合编程实现的并行化优化在算法思想上与 MPI 实现的并行化编程一致，我们只需要在 MPI 进程中再次使用 openMP 优化，以达到进一步任务划分的目的，这样可以在 MPI 的基础上提高并行优化程度。算法思想大体如下:




### 3.4.1 混合编程实现求序列最值程序实现

```
void MaxParallel(int argc,char * argv[],List &L, keyType &max)
{
    max = L.data[0];
    MPI_Status status;
```

```
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size == 1)
{
    max = Max(L);
}
else
{
    if (rank == 0)
    {
        int dest, start, end, length, source, i,temp;
        start = 0;
        length = DATASIZE / size;
        end = start + length;
        for (dest = 1; dest < size; dest++)
        {
            MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            if(start+length<DATASIZE)
                MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            else
                MPI_Send(&L.length, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            start += length;
            end += length;
        }
        #pragma omp parallel for
        for (i = start; i < end; i++)
```

```
        {
            if (L.data[i] > max)
                max = L.data[i];
        }
        for (i = 1; i < size; i++)
        {
            source = i;
            MPI_Recv(&temp, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
            if (max < temp)
                max = temp;
        }
    }
    if (rank > 0)
    {
        int start, end, length, dest,source, i, temp;
        source = 0;
        dest = 0;
        MPI_Recv(&start, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&end, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        #pragma omp parallel for
        for (i = start; i < end; i++)
        {
            temp = L.data[i];
            if (L.data[i] > temp)
                temp = L.data[i];
        }
        MPI_Send(&temp, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
    }
```

```
            MPI_Finalize();
        }
    }
```

### 3.4.2 程序分析

基于 MPI 与 openMP 的求最值程序实现方式我们从上一节代码中可以看到，它与使用基于 MPI 的求最值程序实现方式类似。编写一个 MaxParallel 函数来完成计算任务，该函数有 4 个参数：

argc,argv 参数通过 main 函数传入，以用来初始化 MPI 程序。

List 参数包含有待求最值得序列数据。

Max 参数返回求得的最值。

为了运行该程序，我们需要包含 MPI 标准头文件 mpi.h 和 c 语言输入输出标准头文件 stdio.h。程序开始是以

```
MPI_Init(&argc, &argv);
```

该函数是 MPI 标准接口函数，用来初始化 MPI 运行环境。

接下来我们通过 MPI 的另外两个函数来获取 MPI 进程标识符和 MPI 进程总数。

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

接下来便是程序最重要的一部分，我们首先判断 MPI 进程总数，如果 MPI 进程总数为 1，说明我们不需要使用并行化操作，则直接调用非并行化程序，如果 MPI 进程总数不为 1，则采取并行化方式运行程序。

```
if (size == 1)
{
    max = Max(L);
}
```

接下来我们开始根据 MPI 进程标识符来操作数据，首先如果是在 rank=0 进程中，我们先要求序列 L 进行划分，将其划分为不同的子序列，并将一些必要的划分依据发送给其他 MPI 进程。划分依据为 start,end 下标，也就是说主进程将划分的下标 start,end 两个子序列界限位发送给其他 MPI 进程。

```
MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
if(start+length<DATASIZE)
    MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
else
    MPI_Send(&L.length, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
```

这样其他 MPI 进程负责处理下标从 start 到 end 位置的最值，每个 MPI 进程求出下标从 start 到 end 位置的最值后将最值发送给主 MPI 进程，主 MPI 进程也负责计算一部分子序列的最值。

其他 MPI 进程负责接收序列划分依据，同时在完成自己的计算任务时采用 openMP 的实现方式完成，这样我们便完成了 MPI 与 openMP 的混合编程。

```
#pragma omp parallel for
for (i = start; i < end; i++)
{
    temp = L.data[i];
    if (L.data[i] > temp)
    temp = L.data[i];
}
    MPI_Send(&temp, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
}
```

主进程负责接收其他 MPI 进程发送过来的最值，并求出所有子序列最值中的最值，这样便求出了整个序列 L 的最值。

0 号 MPI 进程接收其他进程求出的字序列最值，并在求出所有子序列最值得最值。

```
for (i = 1; i < size; i++)
{
    source = i;
    MPI_Recv(&temp, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
    if (max < temp)
```

```
            max = temp;
    }
```

## 3.5 并行优化对比分析

我们使用 MPI,openMP,MPI 与 openMP 混合编程实现了对指定序列求最值的并行化处理，在实验对比环节，为验证 MPI,openMP,MPI 与 openMP 混合编程实现并行对指定序列求最值的正确性。我们分别统计一千万，一亿，两亿数据的求最值的时间。

实验环境为 ubuntu12.04 系统,linux 内核版本号为 Linux version 3.8.0-38-generic，CPU 为 intel Xeon(R) E5640@2.4GHz * 16。

### 3.5.1 数据统计

下列表格统计了基于 MPI 的求序列最值并行优化的程序运行时间统计

| MPI 进程数 | 10000000 | 100000000 | 200000000 |
|---|---|---|---|
| 1 | 0.041589(s) | 0.398662(s) | 0.793824(s) |
| 2 | 0.029685(s) | 0.304573(s) | 0.593166(s) |
| 4 | 0.015141(s) | 0.156304(s) | 0.301913(s) |
| 8 | 0.01261(s) | 0.137959(s) | 0.207026(s) |
| 12 | 0.013825(s) | 0.102162(s) | 0.189898(s) |

表 3.1 MPI 实现并行求序列最值时间统计表格

下列表格统计了基于 openMP 的求序列最值并行优化的程序运行时间统计

| opeMP 线程数 | 10000000 | 100000000 | 200000000 |
|---|---|---|---|
| 1 | 0.039745(s) | 0.403846(s) | 0.808967(s) |
| 2 | 0.019983(s) | 0.203407(s) | 0.406974(s) |
| 4 | 0.010203(s) | 0.102773(s) | 0.2081445(s) |
| 8 | 0.009644(s) | 0.077656(s) | 0.160496(s) |
| 12 | 0.00993(s) | 0.067688(s) | 0.129672(s) |

表 3.2 openMP 实现并行求序列最值时间统计表格

下列表格统计了基于 MPI 与 openMP 混合编程的求序列最值的程序运行时间统计

| MPI 进程数为 1 | | | |
|---|---|---|---|
| openMP 线程数 | 10000000 | 100000000 | 200000000 |
| 1 | 0.043009(s) | 0.410402(s) | 0.821884(s) |
| 2 | 0.0233(s) | 0.207061(s) | 0.407662(s) |
| 4 | 0.010407(s) | 0.104491(s) | 0.204646(s) |
| 8 | 0. 009835(s) | 0.089949(s) | 0.192403(s) |

表 3.3.1 MPI 与 openMP 混合编程的求序列最值时间统计表格（进程为 1）

| MPI 进程数为 2 | | | |
|---|---|---|---|
| openMP 线程数 | 10000000 | 100000000 | 200000000 |
| 1 | 0.038398(s) | 0.343163(s) | 0.689016(s) |
| 2 | 0.018049(s) | 0.242017(s) | 0.533349(s) |
| 4 | 0.028573(s) | 0.208514(s) | 0.410678(s) |
| 8 | 0.02665(s) | 0.215003(s) | 0.305869(s) |

表 3.3.2 MPI 与 openMP 混合编程的求序列最值时间统计表格（进程为 2）

| MPI 进程数为 4 | | | |
|---|---|---|---|
| openMP 线程数 | 10000000 | 100000000 | 200000000 |
| 1 | 0.018759(s) | 0.172398(s) | 0.355931(s) |
| 2 | 0.024526(s) | 0.156762(s) | 0.528087(s) |
| 4 | 0.021879(s) | 0.183081(s) | 0.524183(s) |

表 3.3.3 MPI 与 openMP 混合编程的求序列最值时间统计表格（进程为 4）

通过对实验数据的分析比较,我们发现随着 MPI 进程数或 openMP 线程数的增加，程序的运行时间大幅度的减少，由此可以证明并行算法确实比非并行算法在程序运行效率上有着明显的优势。

为方便比对数据，我们通过对实验数据作图分析来对比实验数据。

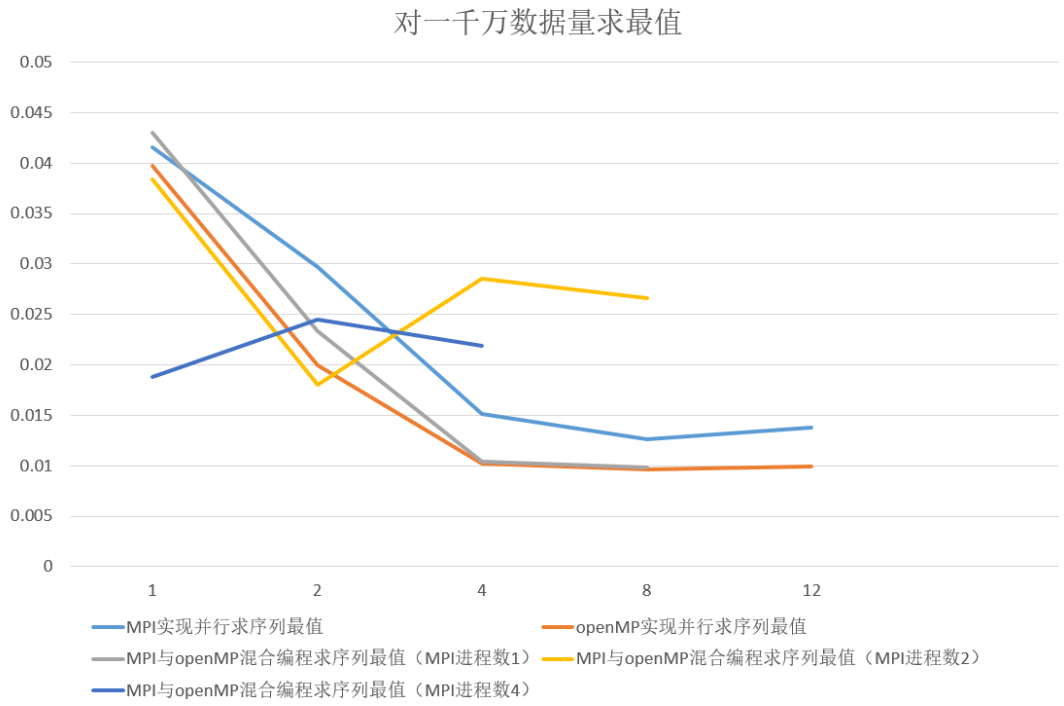



图 3.1 一千万数据量的实验对比图

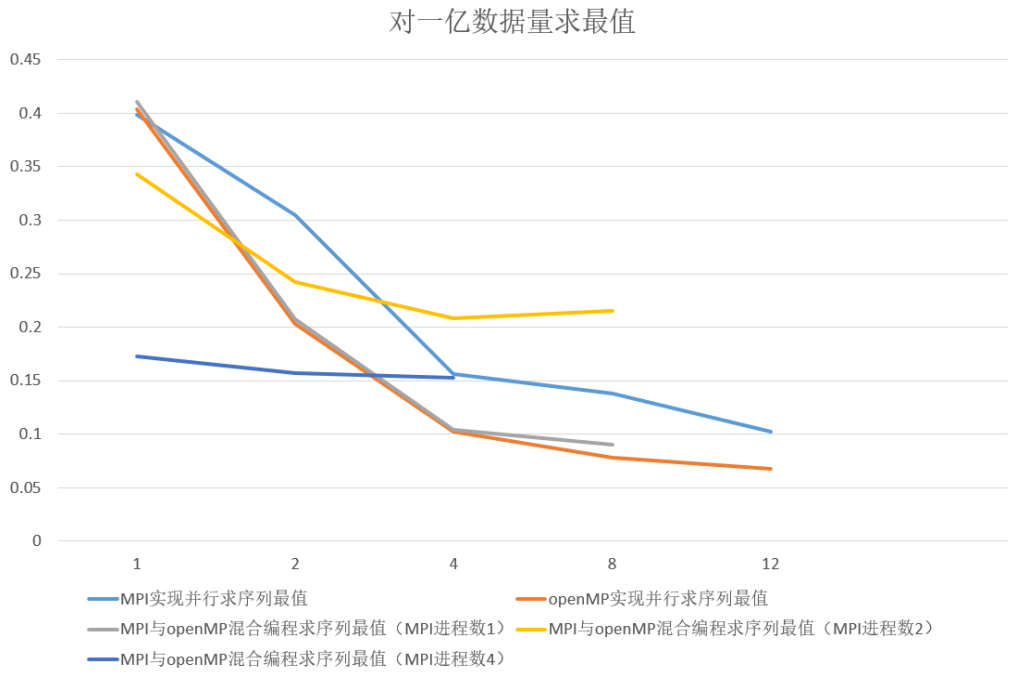



图 3.2 一亿数据量的实验对比图

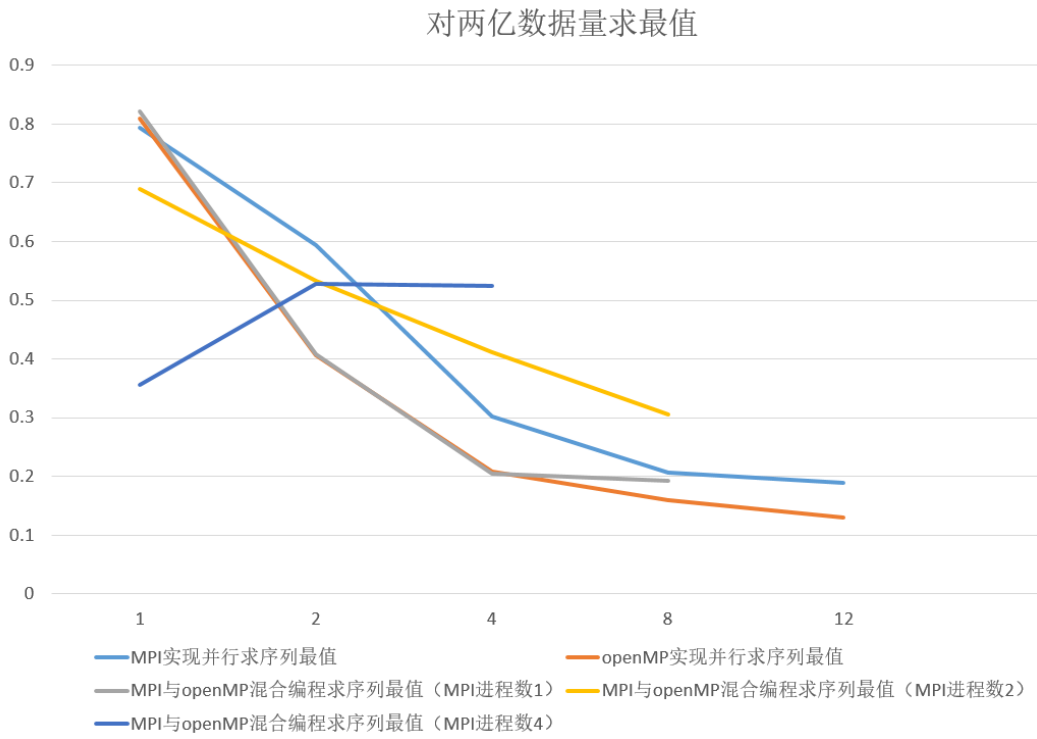



图 3. 3 两亿数据量的实验对比图

### 3. 5. 2 理论分析

首先，我们通过对比单独使用 MPI 与单独使用 openMP 优化程序来分析实验数据。

通过对实验数据的对比分析，我们发现在单独使用 MPI 优化求序列最值时，随着 MPI 进程数的增加，也就是说在并行程度的不断提高下，程序的运行时间有着明显的减少。在单独使用 openMP 实验数据时也可以得出一样的结论。我们发现在实验 openMP 时，由于线程比进程在操作系统资源消耗上比较少，随着 openMP 线程数的不断增加，采用 openMP 比单独采用 MPI 在程序运行时间上有着显著的区别。openMP 在程序的并行优化效果比 MPI 更好，而且随着序列的数据量越大，这个效应更加的明显。

对于这个问题，我们可以很好的解释，因为操作系统对进程的维护比对线程的维护消耗的资源更多，造成程序运行时间的增加。现在我们得出了一个重要结论，在并行求序列最值下我们应该优先采用 openMP 来实现程序，而且随着序列数据量越大 openMP 的效率比 MPI 的效率要好的多。

其次，我们通过对比使用 MPI 与 openMP 混合编程与单独使用 MPI 或单独使用 openMP 来分析实验数据。

通过对实验数据的对比我们发现在进程数相同的情况下使用 MPI 与 openMP 混合编程比单独使用 MPI 编程在程序运行效率上有所不同，使用混合编程时，随着 openMP 线程的增加，使用 MPI 与 openMP 混合编程比单独使用 MPI 编程在时间上更少，使用混合编程时随着 MPI 进程数与 openMP 线程数的增加，程序的运行时间有着明显的减少，程序的运行效率在飞速的提升。

至此，我们可以得出结论在使用 MPI 与 openMP 混合编程时，程序的运行效率比单独使用 MPI 或单独使用 openMP 更好，我们在求序列并行优化时应优先采用 MPI 与 openMP 混合编程模式。

## 3.6 总结

通过对上述实验数据的理论分析，我们可以得出以下几个结论：

第一， MPI 与 openMP 可以并行优化程序，它与非并行化相比，程序的运行时间明显的减少。

第二， 在求序列最值这个问题上，我们应该优先使用 openMP，openMP 比 MPI 在程序执行时间上更少。

第三， 采取 MPI 与 openMP 混合编程比单独使用 MPI 或单独使用 openMP 程序运行时间更少，我们应该优先考虑混合编程模式。

# 第四章　并行排序

## 4.1　非并行排序算法

排序作为计算机中处理数据的常用算法有着非常广泛的应用，排序算法也是其他复杂算法的基础，我们常常为了计算的方便需要先排序然后再去处理数据，排序算法有非常的多种，常用的算法有冒泡排序，插入排序，快速排序等。这些排序算法中有着因为算法本身的问题无法并行化，但其中有的排序算法可以比较容易实现并行化。其中比较简单的算法如枚举排序，枚举排序是一种非常简单的排序算法。我们定义如下序列 L:

$$L=\{a_1,a_2,a_3,\cdots\cdots\cdots a_n\}$$

序列 L 中的数据并不是有序，枚举排序操作如何，定义一个与 L 同大小的操作序列 S, 在 L 中依次比较每个元素比它大的数据个数，这样便可以找到此元素在有序序列 S 中的位置，循环不断的比较。直到找到序列 L 中所有元素在有序序列 S 中的位置，这样便完成了对整个序列 L 的排序操作。

### 4.1.1 非并行排序程序实现

```
void EnumSort(List &L,List &S)
{
    for(int i=0;i<DATASIZE;i++)
    {
        int k=0,data=L.data[i];
        for(int j=0;j<DATASIZE;j++)
        {
            if(data>L.data[j])
            {
                k++;
```

```
            }
        }
        S.data[k]=data;
    }
}
```

4.1.2 程序分析

枚举排序是计算机排序算法中常用的一种排序手段，枚举排序通过对比元素与其他元素的大小关系，最终确定该元素在有序序列中的位置。在 EnumSort 函数中，我们对序列 L 进行枚举排序，并将排好序的有序序列存储在序列 S 中。

```
for(int j=0;j<DATASIZE;j++)
{
    if(data>L.data[j])
    {
        k++;
    }
}
S.data[k]=data;
```

通过计算序列 L 中比 L.data[j]元素大的数据个数，以此来确定 L.data[j]在有序序列 S 中的位置。

通过对程序的运行，我们发现序列 L 中的元素可以通过枚举排序有序的排列在 N 中，算法的时间复杂度为 O(n*n)，看见复杂度为 O(n)。

## 4.2 基于 MPI 的排序优化

MPI 基于消息通讯模型开发的并行接口，我们可以利用 MPI 来实现枚举排序的并行化，依次来达到枚举排序的并行化优化。通过分析非并行化枚举排序，我们可以非常简单的找到并行化的思路，对于给定的序列 L:

$$L=\{a_1,a_2,a_3,\cdots\cdots a_n\}$$

假设 L 中元素个数为 N，MPI 进程个数为 n,则我们可以通过这样的操作来实现并行化，因为枚举排序的思路是通过比较 L 的每个元素与其他的元素，以此发现自己在有序序列中的位置，我们可以知道每个元素与其他元素比较是没有数据依赖的，也就是说它们是不相关的，这样便能实现并行化。将序列 L 划分为含有 N/n 个元素的子序列，每个 MPI 进程负责找到其中子序列元素在有序序列的位置。这样便将枚举排序实现了并行化。

### 4.2.1 基于 MPI 并行排序程序实现

```
void EnumSortParallel(int argc, char *argv[],List &L,List &S)
{
    int rank, size;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size==1)
    {
        EnumSort(L,S);
    }
    else
    {
        if(rank==0)
        {
            double flag;
            int start,end,dest;
            start=0;
            end=DATASIZE/size;
            for(dest=1;dest<size;dest++)
            {
                MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
```

```
            MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            start=start+end;
        }
        for(int i=start;i<DATASIZE;i++)
        {
            int k=0,data=L.data[i];
            for(int j=0;j<DATASIZE;j++)
            {
                if(data>L.data[j])
                {
                    k++;
                }
            }
            S.data[k]=data;
        }
        for(dest=1;dest<size;dest++)
        {
            MPI_Recv(&flag, 1, MPI_INT, dest, 2, MPI_COMM_WORLD, &status);
        }
    }
    if(rank>0)
    {
        int start,end,source;
        source=0;
        MPI_Recv(&start, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&end, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        for(int i=start;i<end;i++)
        {
```

```
                int k=0,data=L.data[i];
                for(int j=0;j<DATASIZE;j++)
                {
                    if(data>L.data[j])
                    {
                        k++;
                    }
                }
                S.data[k]=data;
            }
            MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD);
        }
        MPI_Finalize();
    }
}
```

4. 2. 2 程序分析

基于 MPI 的排序程序实现方式我们从上一节中可以看到，我们通过实现一个 EnumSortParallel 函数来完成计算任务，该函数有 4 个参数：

argc,argv 参数通过 main 函数传入，以用来初始化 MPI 程序。

L 参数包含有待排序的序列数据。

S 参数用来存储排序完成后的序列数据

为了运行该程序，我们必须包含 MPI 标准头文件 mpi.h 和 c 语言输入输出标准头文件 stdio.h。程序开始是以

MPI_Init(&argc, &argv);

该函数是 MPI 标准接口函数，用来初始化 MPI 运行环境。

接下来我们通过 MPI 的另外两个函数来获取 MPI 进程标识符和 MPI 进程总数。

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &size);

接下来便是程序最重要的一部分，我们首先判断 MPI 进程总数，如果 MPI 进程总数为 1，说明我们不需要使用并行化操作，则直接调用非并行化程序，如果 MPI 进程总数不为 1，则采取并行化方式运行程序。

```
if(size==1)
{
    EnumSort(L,S);
}
```

接下来我们开始根据 MPI 进程标识符来操作数据，首先如果是在 rank=0 进程中，我们先要求序列 L 进行划分，将其划分为不同的子序列，并将一些必要的划分依据发送给其他 MPI 进程。划分依据为 start,end 下标，也就是说主进程将划分的下标 start,end 两个子序列界限位发送给其他 MPI 进程。

```
MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
```

这样其他 MPI 进程负责求出下标从 start 到 end 位置的数据在有序序列上的位置，主 MPI 进程也负责求一部分数据在有序序列上的位置，这样便求出了整个序列 L 上的所有元素在有序序列上的位置。

```
MPI_Recv(&start, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
MPI_Recv(&end, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
for(int i=start;i<end;i++)
{
    int k=0,data=L.data[i];
    for(int j=0;j<DATASIZE;j++)
    {
        if(data>L.data[j])
            k++;
    }
    S.data[k]=data;
}
```

## 4.3 基于 openMP 的排序优化

前面我们使用 MPI 实现了枚举排序的并行化优化，对于枚举排序如何并行化的思路也做出了介绍，现在我们使用 openMP 来实现枚举排序的并行化。使用 openMP 来实现枚举排序的并行化比使用 MPI 更加的简单，在思路上和使用 MPI 处理一样，只不过 openMP 是基于线程级别上的并行化处理。

### 4.3.1 基于 openMP 并行排序程序实现

```
void EnumSort(List &L,List &S)
{
    #pragma omp parallel private(i,j,k)
    {
        #pragma omp for
        for(int i=0;i<DATASIZE;i++)
        {
            int k=0,data=L.data[i];
            for(int j=0;j<DATASIZE;j++)
            {
                if(data>L.data[j])
                {
                    k++;
                }
            }
            S.data[k]=data;
        }
    }
}
```

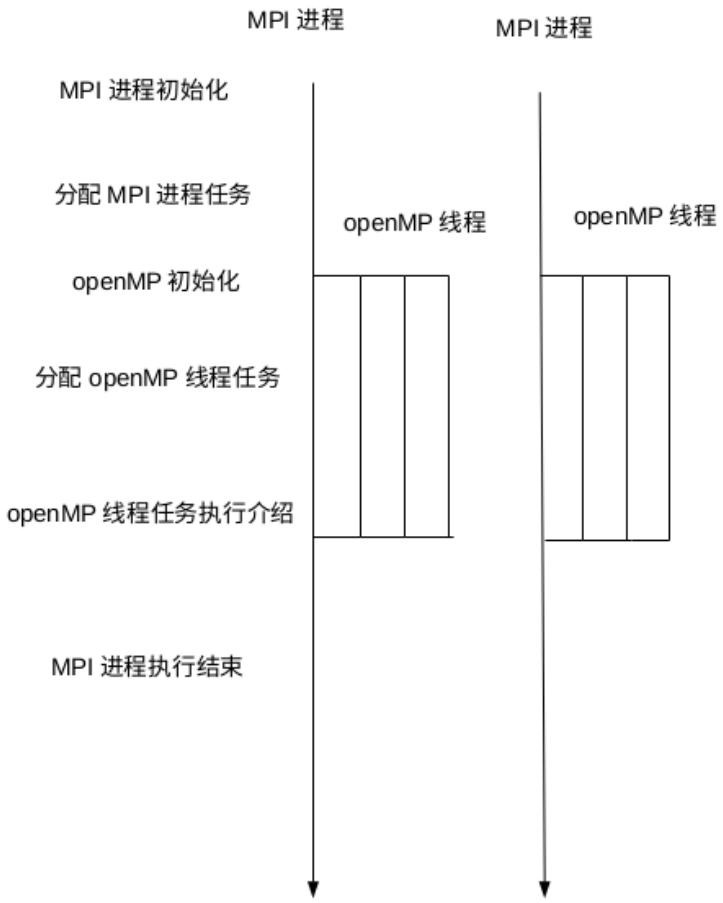
4.3.2 程序分析

openMP 采用编译指令的方式来进行并行优化，它的操作十分的简单，与非并行化程序在程序结构上十分相似，为了实现 openMP 并行化，我们只需要在可以并行化的代码块上加入 openMP 编译指令#pragma omp parallel private(i,j,k) 。同时在可以并行化的 for 循环上加上 openMP 编译指令#pragma omp for。

```
#pragma omp for
for(int i=0;i<DATASIZE;i++)
{
    int k=0,data=L.data[i];
    for(int j=0;j<DATASIZE;j++)
    {
        if(data>L.data[j])
        {
            k++;
        }
    }
    S.data[k]=data;
}
```

## 4.4 基于 MPI 与 openMP 混合编程的排序优化

前面我们通过使用 MPI 与 openMP 混合编程实现了矩阵乘法，求序列最值的问题，对于 MPI 与 openMP 如何混合编程做出了许多的介绍，我们知道 MPI 是基于进程实现的并行化，openMP 则是基于线程的并行化，两者在本质上有着非常大的区别，对于给定的需要并行化问题，我们先分析问题是否可以并行化处理，这样我们先对问题在进程级别上进行任务划分，也就是说使用 MPI 来实现并行化，其次对于每个进程分配到的任务，我们又可以在线程的级别上对任务再次划分,也就是说使用 openMP 编程，这样便可以进一步提高并行化力度，提高了并行化的效率。




在前面的介绍中我们便使用了这张图来形象的分析出了 MPI 与 openMP 混合编程的原理，已经如何使用代码来具体编程实现。

### 4.4.1 基于混合编程的并行排序程序实现

```
void EnumSortParallel(int argc, char *argv[],List &L,List &S)
{
    int rank, size;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size==1)
    {
```

```
        EnumSort(L,S);
    }
    else
    {
        if(rank==0)
        {
            double flag;
            int start,end,dest;
            start=0;
            end=DATASIZE/size;
            for(dest=1;dest<size;dest++)
            {
                MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
                MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
                start=start+end;
            }
              #pragma omp parallel private(i,j,k)
              {
                  #pragma omp for
               for(int i=start;i<DATASIZE;i++)
               {
                    int k=0,data=L.data[i];
                    for(int j=0;j<DATASIZE;j++)
                    {
                        if(data>L.data[j])
                        {
                            k++;
                        }
```

```
                }
                S.data[k]=data;
            }
        }
    for(dest=1;dest<size;dest++)
    {
        MPI_Recv(&flag, 1, MPI_INT, dest, 2, MPI_COMM_WORLD, &status);
    }
}
if(rank>0)
{
    int start,end,source;
    source=0;
    MPI_Recv(&start, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&end, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
      #pragma omp parallel private(i,j,k)
      {
          #pragma omp for
        for(int i=start;i<end;i++)
        {
            int k=0,data=L.data[i];
            for(int j=0;j<DATASIZE;j++)
            {
                if(data>L.data[j])
                {
                    k++;
                }
            }
```

```
                    S.data[k]=data;
                }
            }
            MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD);
        }
        MPI_Finalize();
    }
}
```

### 4.4.2 程序分析

基于 MPI 与 openMP 混合编程的排序程序实现方式我们从上一节中可以看到，我们通过实现一个 EnumSortParallel 函数来完成计算任务，该函数有 4 个参数：

argc,argv 参数通过 main 函数传入，以用来初始化 MPI 程序。

L 参数包含有待排序的序列数据。

S 参数用来存储排序完成后的序列数据

为了运行该程序，我们必须包含 MPI 标准头文件 mpi.h 和 c 语言输入输出标准头文件 stdio.h。程序开始是以

```
MPI_Init(&argc, &argv);
```

该函数是 MPI 标准接口函数，用来初始化 MPI 运行环境。

接下来我们通过 MPI 的另外两个函数来获取 MPI 进程标识符和 MPI 进程总数。

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

接下来便是程序最重要的一部分，我们首先判断 MPI 进程总数，如果 MPI 进程总数为 1，说明我们不需要使用并行化操作，则直接调用非并行化程序，如果 MPI 进程总数不为 1，则采取并行化方式运行程序。

```
if(size==1)
{
    EnumSort(L,S);
}
```

接下来我们开始根据 MPI 进程标识符来划分数据，首先如果是在 rank=0 进程中，我们先要求序列 L 进行划分，将其划分为不同的子序列，并将一些必要的划分依据发送给其他 MPI 进程。划分依据为 start,end 下标，也就是说主进程将划分的下标 start,end 两个子序列界限位发送给其他 MPI 进程。

```
MPI_Send(&start, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
MPI_Send(&end, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
```

这样其他 MPI 进程负责求出下标从 start 到 end 位置的数据在有序序列上的位置，主 MPI 进程也负责求一部分数据在有序序列上的位置，这样便求出了整个序列 L 上的所有元素在有序序列上的位置。

其他 MPI 进程负责接收划分依据，并行执行自己的计算任务。

```
MPI_Recv(&start, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
MPI_Recv(&end, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
```

相比于单独使用 MPI 编程，使用 MPI 与 openMP 混合编程的特点是在各自 MPI 进程执行自己的计算任务时应该使用 openMP 来实现。

```
#pragma omp parallel private(i,j,k)
{
    #pragma omp for
    for(int i=start;i<end;i++)
    {
        int k=0,data=L.data[i];
        for(int j=0;j<DATASIZE;j++)
        {
            if(data>L.data[j])
            {
                k++;
            }
        }
        S.data[k]=data;
```

```
        }
        MPI_Send(&source, 1, MPI_INT, source, 2, MPI_COMM_WORLD);
    }
```

## 4.5 并行优化对比分析

我们使用 MPI,openMP 和 MPI 与 openMP 混合编程实现了枚举排序的并行化优化，在实验对比环境我们分别对一万，十万，二十万数据排序，分别记录 MPI,openMP 和 MPI 与 openMP 混合编程实现枚举排序的程序运行时间。

实验环境为 ubuntu12.04 系统,linux 内核版本号为 Linux version 3.8.0-38-generic，CPU 为 intel Xeon(R) E5640@2.4GHz*16。

### 4.5.1 数据统计

下列表格统计了基于 MPI 的排序并行优化的程序运行时间统计

| MPI 进程数 | 10000 | 100000 | 200000 |
|---|---|---|---|
| 1 | 0.721356(s) | 71.637202(s) | 287.400746(s) |
| 2 | 0.374576(s) | 36.501342(s) | 145.581476(s) |
| 4 | 0.187963(s) | 18.135853(s) | 72.683338(s) |
| 8 | 0.136146(s) | 9.524575(s) | 49.026629(s) |
| 12 | 0.095184(s) | 9.49021(s) | 28.500561(s) |

表 4.1 MPI 实现并行排序时间统计表格

下列表格统计了基于 openMP 的排序并行优化的程序运行时间统计

| openMP 线程数 | 10000 | 100000 | 200000 |
|---|---|---|---|
| 1 | 0.496374(s) | 49.006654(s) | 196.089782(s) |
| 2 | 0.254453(s) | 24.672865(s) | 98.761748(s) |
| 4 | 0.153096(s) | 13.319548(s) | 52.809(s) |
| 8 | 0.154(s) | 12.100951(s) | 42.995844(s) |
| 12 | 0.139337(s) | 8.375999(s) | 34.524425(s) |

表 4.2 openMP 实现并行排序时间统计表格

下列表格统计了基于 MPI 与 openMP 混合编程的排序并行优化的程序运行时间统计

| MPI 进程数为 1 | | | |
|---|---|---|---|
| openMP 线程数 | 10000 | 100000 | 200000 |
| 1 | 0.73348(s) | 73.077974(s) | 291.621681(s) |
| 2 | 0.741072(s) | 72.672775(s) | 254.67655(s) |
| 4 | 0.738113(s) | 73.144233(s) | 246.283728(s) |
| 8 | 0.73147(s) | 71.155234(s) | 244.342323(s) |

表 4.3.1 MPI 与 openMP 混合编程实现并行排序时间统计表格(进程数为 1)

| MPI 进程数为 2 | | | |
|---|---|---|---|
| openMP 线程数 | 10000 | 100000 | 200000 |
| 1 | 0.362523(s) | 36.503034(s) | 144.745343(s) |
| 2 | 0.387337(s) | 35.678754(s) | 145.597403(s) |
| 4 | 0.368061(s) | 36.768768(s) | 142.443343(s) |
| 8 | 0.366574(s) | 34.876667(s) | 134.6454845(s) |

表 4.3.2 MPI 与 openMP 混合编程实现并行排序时间统计表格(进程数为 2)

| MPI 进程数为 4 | | | |
|---|---|---|---|
| openMP 线程数 | 10000 | 100000 | 200000 |
| 1 | 0.199411(s) | 18.132426(s) | 75.324243(s) |
| 2 | 0.182659(s) | 17.787658(s) | 73.465047(s) |
| 4 | 0.205902(s) | 16.765567(s) | 72.618292(s) |

表 4.3.3 MPI 与 openMP 混合编程实现并行排序时间统计表格(进程数为 4)

通过对实验数据的分析比较,我们发现随着 MPI 进程数或 openMP 线程数的增加，程序的运行时间大幅度的减少，由此可以证明并行算法确实比非并行算法在程序运行效率上有着明显的优势。

为方便比对数据，我们通过对实验数据作图分析来对比实验数据。

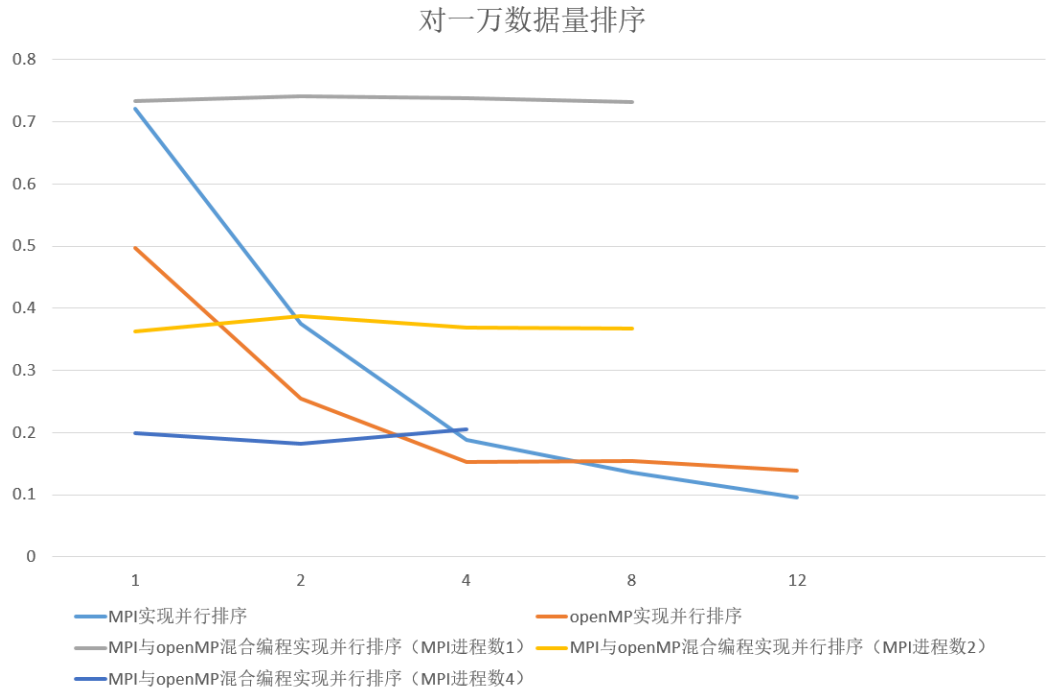



图 4.1 一万数据排序对比图

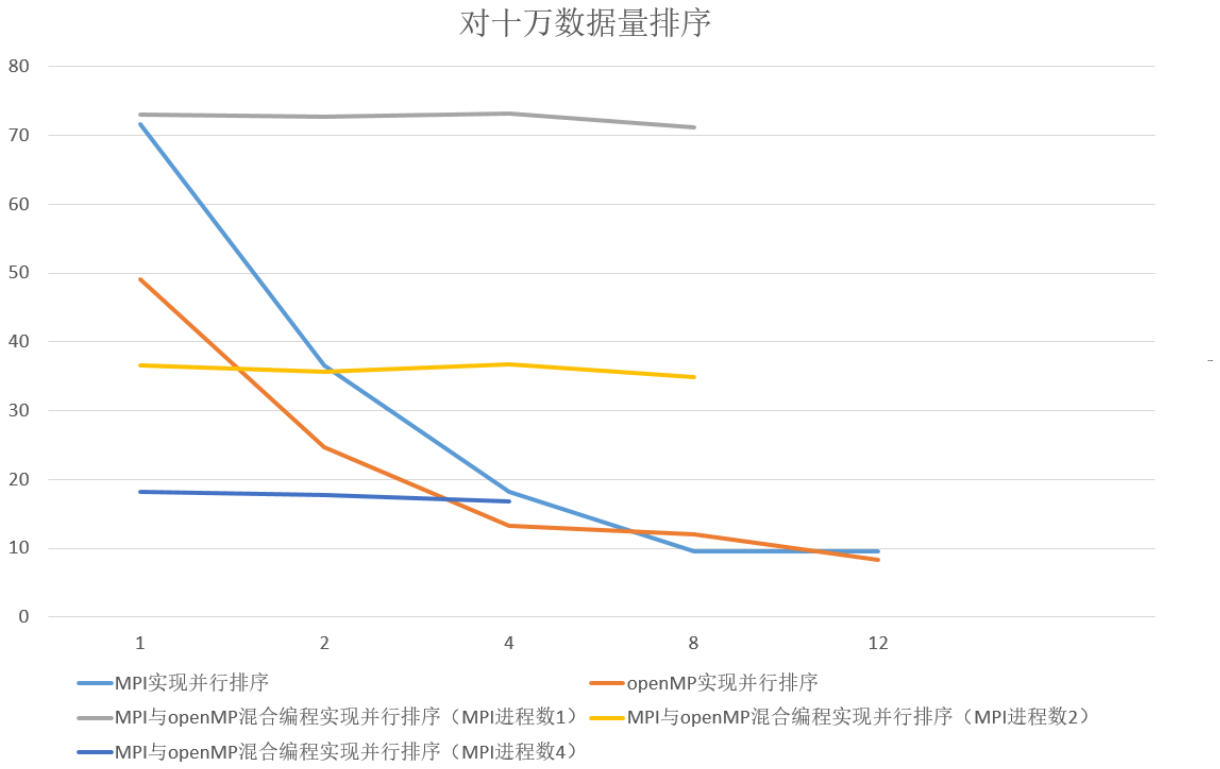



图 4.2 十万数据排序对比图

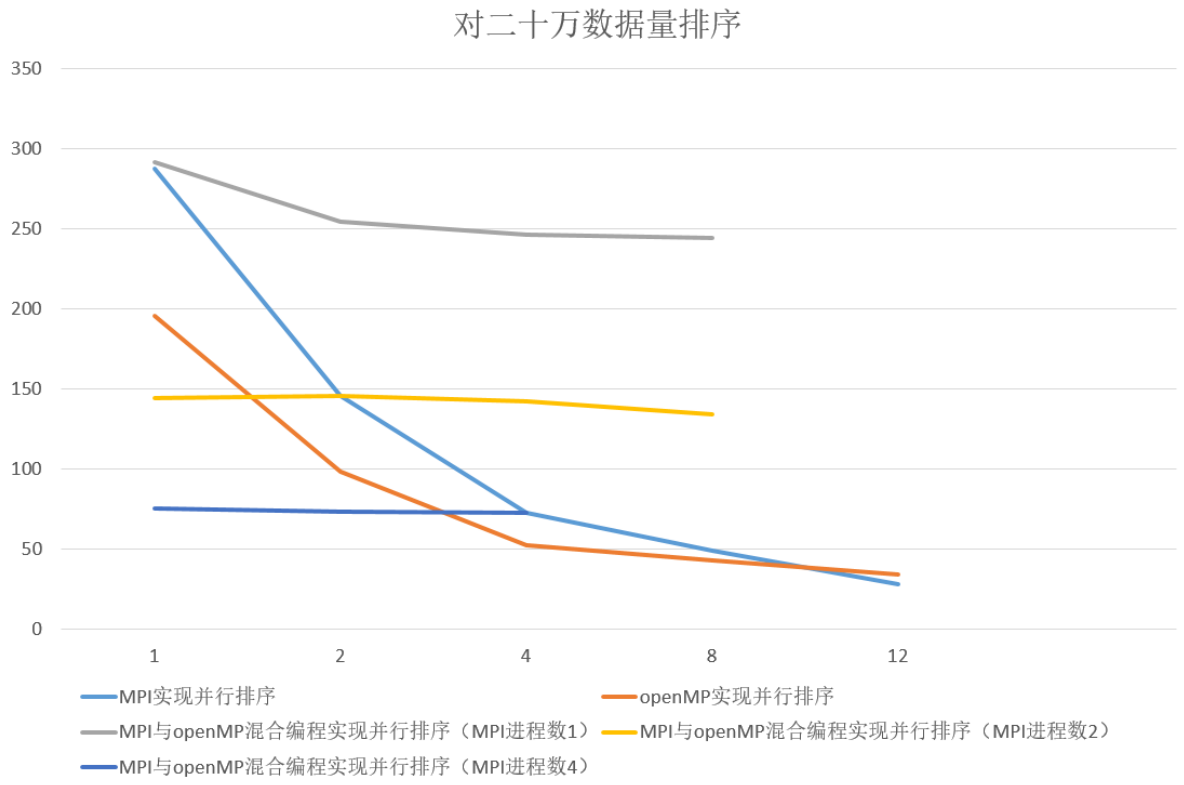



图 4.3 二十万数据排序对比图

### 4.5.2 理论分析

首先，我们通过对比单独使用 MPI 与单独使用 openMP 优化程序来分析实验数据。

通过对实验数据的对比分析，我们发现在单独使用 MPI 优化排序时，随着 MPI 进程数的增加，也就是说在并行程度的不断提高下，程序的运行时间有着明显的减少。在单独使用 openMP 实验数据时也可以得出一样的结论。我们发现在实验 openMP 时，由于线程比进程在操作系统资源消耗上比较少，随着 openMP 线程数的不断增加，采用 openMP 比单独采用 MPI 在程序运行时间上有着显著的区别。openMP 在程序的并行优化效果比 MPI 更好，而且随着序列的数据量越大，这个效应更加的明显。

对于这个问题，我们可以很好的解释，因为操作系统对进程的维护比对线程的维护消耗的资源更多，造成程序运行时间的增加。现在我们得出了一个重要结论，在并行排序下我们应该优先采用 openMP 来实现程序，而且随着序列数据量越大 openMP 的效率比 MPI 的效率要好的多。

其次，我们通过对比 MPI 与 openMP 混合编程与单独使用 MPI 或单独使用 openMP 来分析。

我们从 MPI 与 openMP 混合编程的实验数据中发现，在进程不变的情况下，随着 openMP 线程的不断增加，程序在运行效率上提升的不是很明显，但是随着排序的数据量不断的增加，程序在运行效率上有所提升，我们从中可以发现，并不是所有的情况下都适合混合编程，在数据量不大的前提下，混合编程对程序的性能提升不是很明显，有时反而会比单独使用 MPI 或单独使用 openMP 在程序运行时间上反而更多。

## 4.6 总结

通过对上述实验数据的理论分析，我们可以得出以下几个结论：

第一，MPI 与 openMP 可以并行优化程序，它与非并行化相比，程序的运行时间明显的减少。

第二，在排序这个问题上，我们应该优先使用 openMP，openMP 比 MPI 在程序执行时间上更少。

第三， 在排序的数据量不大的前提下，混合编程对并行程序的性能提升不明显，我们在此时应该优先采用 MPI 或采用 openMP 技术。

第四， 在排序的数据量很大时，混合编程对并行程序的性能提升开始显现，此时我们应该优先考虑 MPI 与 openMP 混合编程来优化排序。

# 第五章　测试工具

## 5.1　项目代码介绍

整个毕业设计在 linux 环境下面完成，编译工具为 g++，文本编辑工具 gedit，调试工具为 gdb,代码组织列表如下:

```
Design
├── Matrix
│   ├── data.txt
│   ├── Matrix.cpp
│   ├── Matrix.sh
│   ├── myMatrix.h
│   ├── Omp_Matrix.cpp
│   ├── Omp_MPI_Matrix.cpp
│   └── Random.cpp
├── Max
│   ├── data.txt
│   ├── Max.cpp
│   ├── Max.sh
│   ├── myMax.h
│   ├── Omp_Max.cpp
│   ├── Omp_MPI_Max.cpp
│   └── Random.cpp
├── readme.txt
├── run.sh
├── Sort
│   ├── data.txt
│   ├── mySort.h
│   ├── Omp_MPI_Sort.cpp
│   ├── Omp_Sort.cpp
│   ├── Random.cpp
│   ├── Sort.cpp
│   └── Sort.sh
└── UI
    ├── exec.sh
    ├── Main.png
    ├── result.txt
    ├── UI.c
    ├── UI.glade~
    ├── UI.xml~
    └── wait.gif
```

图 5.1 项目工程图

以下是各个文件的解释说明:

Design 项目主目录

run.sh 项目运行脚本

readme.txt 帮助文件

Matrix 实现 MPI,openMP,MPI 与 openMP 混合编程的并行矩阵乘法主目录

data.txt 用来保存用来计算的矩阵数据

myMatrix.h 头文件中定义了矩阵结构体和一些编译条件指令

Random.cpp 为产生随机矩阵源文件

Matrix.cpp 为 MPI 实现并行矩阵乘法源文件

Omp_Matrix.cpp 为 openMP 实现并行矩阵乘法源文件

Omp_MPI_Matrix.cpp 为 MPI 与 openMP 混合编程实现并行矩阵乘法源文件

Matrix.sh 为 linux 下测试脚本

Max 实现 MPI,openMP,MPI 与 openMP 混合编程的求序列最值主目录

data.txt 用来保存序列数据

myMax.h 头文件中定义了序列结构体和一些编译条件指令

Random.cpp 为产生随机序列数据源文件

Max.cpp 为 MPI 实现并行求序列最值源文件

Omp_Max.cpp 为 openMP 实现并行求序列最值源文件

Omp_MPI_Max.cpp 为 MPI 与 openMP 混合编程实现并行求序列最值源文件

Matrix.sh 为 linux 下测试脚本

Sort 实现 MPI,openMP,MPI 与 openMP 混合编程的并行排序主目录

data.txt 用来保存序列数据

mySort.h 头文件中定义了序列结构体和一些编译条件指令

Random.cpp 为产生随机序列数据源文件

Sort.cpp 为 MPI 实现并行排序源文件

Omp_Sort.cpp 为 openMP 实现并行排序源文件

Omp_MPI_Sort.cpp 为 MPI 与 openMP 混合编程实现并行排序源文件

Matrix.sh 为 linux 下测试脚本

UI 图形界面测试工具主目录

result.txt 用来保存实验结果

exec.sh 测试脚本

Main.png 图形界面图标文件

UI.c 图形界面测试工具源文件

Wait.gif 图形界面资源文件

具体代码源文件见附录

## 5.2 基于 GTK 图形界面测试工具

GTK 是一套源码以 LGPL 许可协议分发、跨平台的图形工具包。GTK 是一个功能强大、设计灵活的通用图形库，是 Linux 下开发图形界面的应用程序的主流开发工具之一。并且，GTK 也支持 Windows 与 Mac OS 系统，使用 GTK 开发出的图形界面可以跨平台编译。

为了测试程序的方便，我使用 GTK 编写了一个图形界面的测试工具，测试工具界面如下:


图 5.2 测试工具界面

在文件选项中可以选择打开指定的文件，为了测试的方便，同时编写了一个测试脚本exec.sh,只要打开该文件即可，在类型选项中定义了如下的输入选项:

文件打开界面:

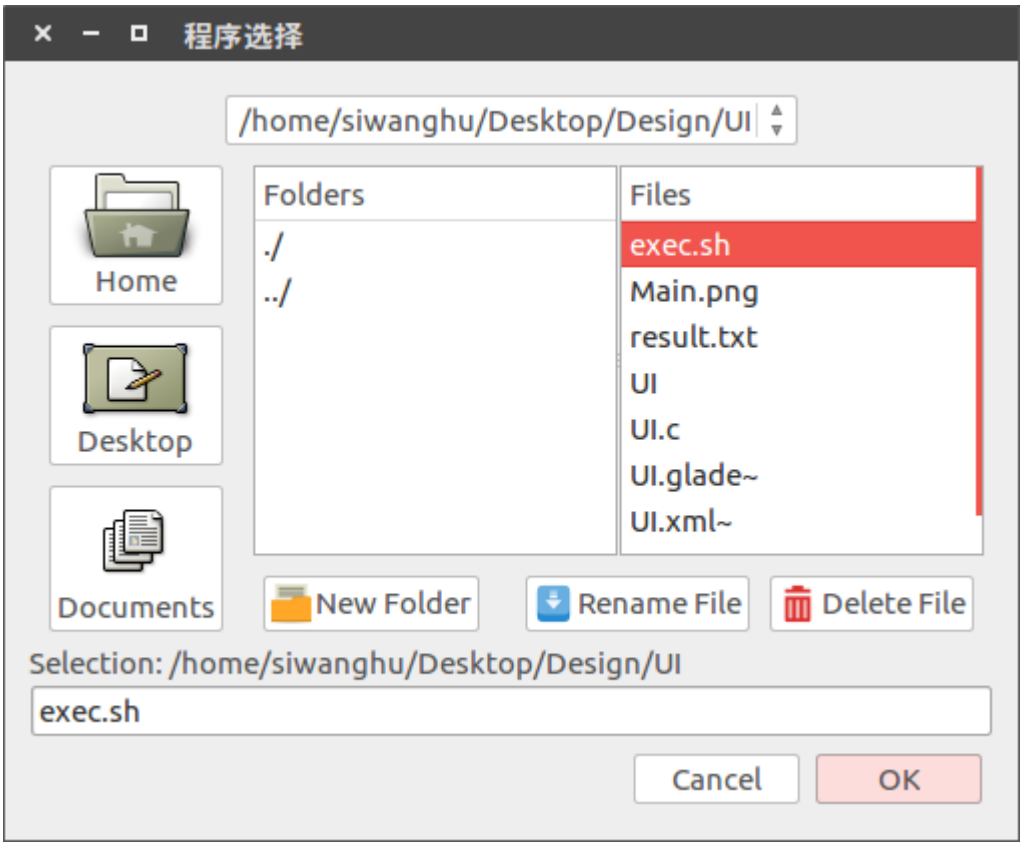



图 5.3 文件打开界面

类型提示界面:


图 5.4 类型提示界面


图 5.5 程序运行界面

测试工具运行结果界面：


图 5.6 运行结果界面

# 第六章　总结与展望

## 6.1　总结与展望

随着科学家与工程师对并行计算的不断探索，并行计算得到了飞速的发展，现在已经广泛的应用到了科学研究与工程领域，尤其在工程领域应用广泛。程序员将传统的非并行算法程序通过改进，使其转换为并行程序，以此来提高计算机的计算效率，大大的减少了程序的执行时间，并行编程也以成为了程序员编写复杂计算问题优先考虑的解决方式。可以这样说，并行计算已经得到了广泛的应用。

毕业设计主要任务是使用 MPI 与 openMP 混合编程实现矩阵乘法，求序列最值，排序的并行化处理。通过使用 MPI,openMP,MPI 与 openMP 混合编程来对比并行程序与非并行程序的效率，以此来验证并行计算可以提高计算机的计算效率，减少程序的运行时间。通过这次对并行编程的研究使我加深了对并行计算的理解，同时也对并行编程有了深刻的认识。对于矩阵乘法的并行化处理通过对矩阵的划分，将矩阵乘法的问题分配给不同的进程处理，这样便可以实现矩阵乘法的并行化处理，试验数据证明，这种并行处理方式是正确可行的。对于求指定序列最值的问题，通过对序列的划分，不同的进程求各自子序列的最值，最后在求所有子序列最值的最值，最终便可以求出整个序列的最值，对于枚举排序问题，通过分析排序的操作处理，对枚举排序进行并行化处理，也可以实现枚举排序的并行化。

通过这次毕业设计的完成，我深刻体会到并行计算在计算机领域的重要性。完成毕业设计的过程中，虽然面临着许多的问题，但是在自己不断的探索下，都得到了解决。这次过程使我对科学研究产生了浓厚的兴趣。

# 参 考 文 献

[1] 罗省贤，何大可．基于 MPI 的网络并行计算环境及应用[M]．西南交通大学出版社，2001.

[2] 吕捷，张天序，张必银．MPI 并行计算在图像处理方面的应用[J]．红外与激光工程，2004，33(5):496-499.

[3] 崔焕庆，吴哲辉，韩丛英. MPI 通信函数的增广 Petri 网模型[J]．系统仿真学报，2003，15(z1):26-28.

[4] 游佐勇．OpenMP 并行编程模型与性能优化方法的研究及应用[D]．成都理工大学，2011.

[5] 陈辉，孙雷鸣，李录明,等．基于 MPI+OpenMP 的多层次并行偏移算法研究[J]．成都理工大学学报(自科版)，2010，37(5):528-534.

[6] 刘轶，郑守淇，钱德沛．一种分布式共享存储系统的线程分配算法[J]．计算机研究与发展，2000，37(5):521-526.

[7] 李小卫，罗省贤．基干 MPI 的并行 I/O 方法[J]．微型机与应用，2003，22(3).

[8] 李永旭．基于 MPI 标准的并行计算平台的设计与实现[D]．东北师范大学，2007.

[9] 罗秋明，王梅，雷海军．基于 MPI 的匹配方体并行计算研究[J]．计算机应用，2006，26(8):1916-1918.

[10] 罗省贤，何大可．基于 MPI 的网络并行计算环境及应用[M]．西南交通大学出版社，2001.

[11] 刘凯，寇正．OpenMP 在并行计算中的应用[J]．微型机与应用，2003，22(12):12-14.

[12] Smith L, Bull M. Development of mixed mode MPI / OpenMP applications[J]. Scientific Programming, 2001, 9(2-3):83-98.

[13] Karniadakis G M, Kirby R M. Parallel scientific computing in C++ and MPI :[M]. Cambridge University Press, 2003.

[14] Chow E. Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters[J]. 2001.

[15] 杨灿群, 杨学军, 易会战. 扩展双精度浮点并行计算:MPI 方法[J]. 计算机工程与科学, 2010, 32(12):98-101.

[16] 付晓东, 盛谦, 张勇慧. 基于 OpenMP 的非连续变形分析并行计算方法[J]. 岩土力学, 2014(8):2401-2407.

[17] 肖永浩, 莫则尧. 接触问题的 MPI+OpenMP 混合并行计算[J]. 振动与冲击, 2012, 31(15):36-40.

[18] 林荫, 黑保琴. 基于 MPI+OpenMP 混合模型的并行处理算法设计[C]// 中国空间科学学会空间探测专业委员会学术会议. 2008.

[19] 陈国良. 并行计算.第 2 版[M]. 高等教育出版社, 2003.

[20] 计永昶, 丁卫群, 陈国良,等. 一种实用的并行计算模型[J]. 计算机学报, 2001, 24(4):437-441.

# 致　　谢

通过这六个月来的忙碌和学习，这次毕业设计已接近尾声，作为一个即将毕业的大学生，这次经历让我感触很深，由于自身水平的匮乏，难免会有许多考虑不周到的地方，但在老师细心的督促和指导下，使我在这个过程中水平飞速成长，在这里我衷心感谢指导老师，以及一起学习的同学们的帮助，让我能够成功的完成了这次毕业设计。

在毕业设计完成的过程中，我面临着许多的困难，开始接到任务书时，我对于并行计算是完成不知道的，对于如何使用 MPI 和 openMP 编程也一无所知，但是在我的指导老师陈哲老师的悉心帮助下，我从无从下手到开始入门，可以说没有老师的指导我很难下手。在具体毕业设计编程环节中，陈哲老师的研究生陈韬学长给我许多的建议，在编程上给我许多的思路，使我可以及时的把理论研究具体到代码实现上，在这里我衷心的感谢陈韬学长。在毕业设计论文的撰写环节中，我的同学们给了我许多宝贵的建议，我一开始并不知道如何撰写毕业设计论文，在同学们的指导和帮助下，我独立的完成了整个论文的撰写，在这里我也要感谢和我一起同窗读书四年的同学们，也预祝他们今后生活幸福，能成为积极有为的祖国人才。

大学四年的学习时光很快就要过去了，在此我想对我的母校南京航空航天大学表达我由衷的谢意，感谢母校对我的培养，祝母校南京航空航天大学蓬勃发展。

# 附 录

MPI 常用函数说明

int MPI_Init(int *argc, char * argv[])

MPI_Init 函数是 MPI 程序的第一个调用，它完成 MPI 程序所有的初始化工作，所有的 MPI 程序并行部分的第一条可执行语句都是这条语句，这条语句标志着程序并行部分的开始。

int MPI_finalize()

MPI_Finalize 函数是 MPI 程序的最后一个调用，它结束 MPI 程序的运行。他是 MPI 程序的最后一条可执行语句，她标志着并行程序的结束。

int MPI_Comm_rank(MPI_Comm comm ,int* rank)

MPI_Comm_rank 函数调用通过指针返回调用该函数的进程在给定的通信域中的进程标识号。有了这一标识号，不同的进程就可以将自身和其他的进程区别开来，实现各进程的并行和协作。

int MPI_Comm_size(MPI_Comm comm ， int* size)

MPI_Comm_size 函数调用返回给定的通信域中所包括的进程的总个数，不同的进程通过这一调用得知在给定的通信域中一共有多少个进程在并行执行。

int MPI_Send(void* buf , int cout , MPI_Datatype datatype , int dest , int tag , MPI_Comm comm)

buf 发送缓冲区的起始地址（可选类型）

count 将发送的数据个数（非负整数）

datatype 发送数据的数据类型（句柄）

dest 目的进程标识号（整型）

tag 消息标志（整型）

comm 通信域（句柄）

int MPI_Recv(void* buf , int count , MPI_Datatype datapyte , int source , int tag , MPI_Comm comm , MPI_Status* status)

buf 接收缓冲区的起始地址（可选类型）

count 最多接收的数据个数（非负整数）

datatype 接收数据的数据类型（句柄）

source 接收数据的来源进程标识号（整型）

tag 消息标识与相应的发送操作的表示相匹配（整型）

comm 本进程和发送进程所在的通信域（句柄）

status 返回状态（状态类型）