

**多路处理器计算机教学实验系统**

# **《计算机系统结构》**

## **实验指导书**

深圳大学计算机与软件学院

国家高性能计算中心深圳分中心



# 目 录

第一部分 实验环境介绍.....	4
1 实验平台系统简介.....	4
2 操作系统安装.....	7
第二部分 教学实验指导.....	13
第三部分 实验操作指导.....	155
实验 1 Amdahl 定量原理.....	15
实验 2 循环展开与指令流水.....	20
实验 3 LLC 对整体效率的影响.....	24
实验 4 cache 乒乓效应实验.....	31
实验 5 访存行为对 cache 性能的影响.....	37
实验 6 cache 层次及容量评估.....	40
实验 7 TLB 命中与缺失对访存性能的影响.....	44
实验 8 内存控制器带宽与竞争.....	47
实验 9 访存中的 NUMA 因素.....	51
实验 10 集群结构体验——MPI 编程.....	错误！未定义书签。
附录 1 实验报告格式.....	64

# 第一部分 实验环境介绍

## 1 实验平台系统简介

多路处理器计算机教学实验系统是由龙芯 3A 处理器构成的、内可配置外可扩展结构的实验硬件平台。该实验平台由多路处理器教学实验箱和一台文件系统服务器组成。多路处理器教学实验箱上有对称的 4 个处理单元，每个处理单元包含一颗龙芯 3A 四核处理器。4 个处理单元既可通过网络互连为多处理机集群架构，也可通过 HT 总线互连为 CC-NUMA 架构。并且多个实验平台可通过网络和 HT 实现更多的处理器互连。文件系统服务器用来统一存放 NFS 系统，为处理单元提供内核和文件系统。在教学中，教师办公用机即可配置为文件系统服务器。实验平台的基本结构和软硬件结构图如下所示。

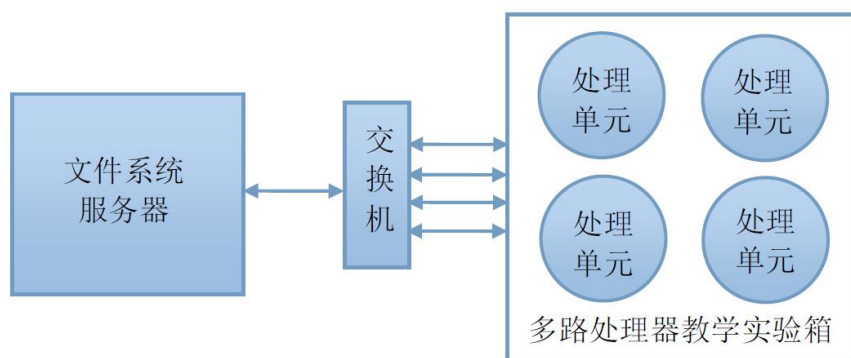


图 1-1 实验平台基本结构图

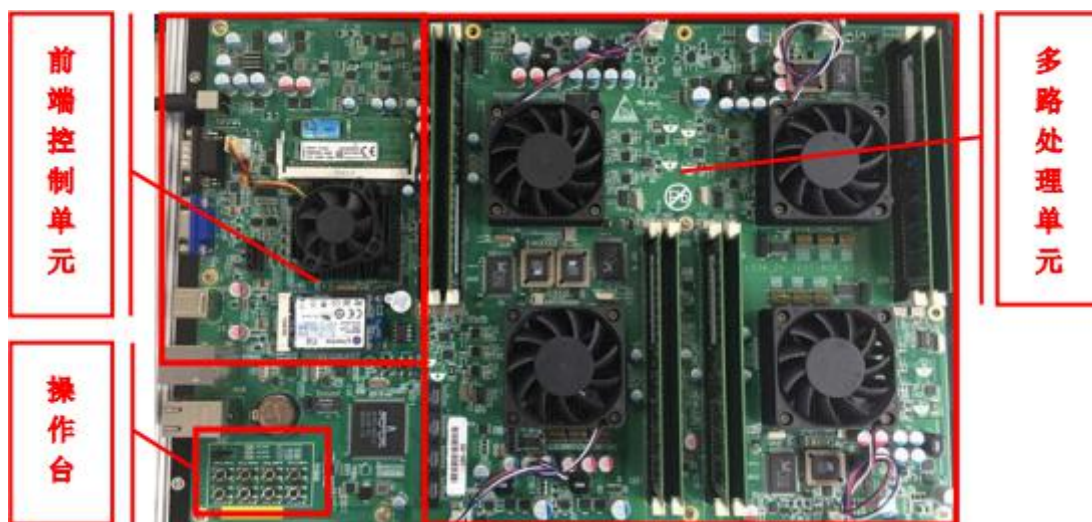


图 1-2 多路处理器教学实验箱实物图

多路处理器教学实验箱由处理板和控制板两部分组成，整体实物如图 1-2 所

示。处理板是实验平台的主要部分，可运行多处理器程序，实现并行程序的设计与开发。控制板主要负责把处理板上的网络、串口等资源扩展成标准接口，同时负责给处理板供电和进行相应的控制。

### 1.1. 处理板

处理板承载 4 个处理单元，编号如图 1-3 所示，每个处理单元包括一颗龙芯 3A 四核处理器、DDR2 内存（内存数量和单条容量可根据需要进行配置）、RTL8110 千兆以太网卡芯片、BIOS Flash、串口收发芯片以及电源变换电路等。四个龙芯 3A 处理器在处理板上通过 HT 总线实现互连。另外，通过以太网四个处理单元还可以在板外进行互连。处理器板的结构示意图如图 1-3 所示。

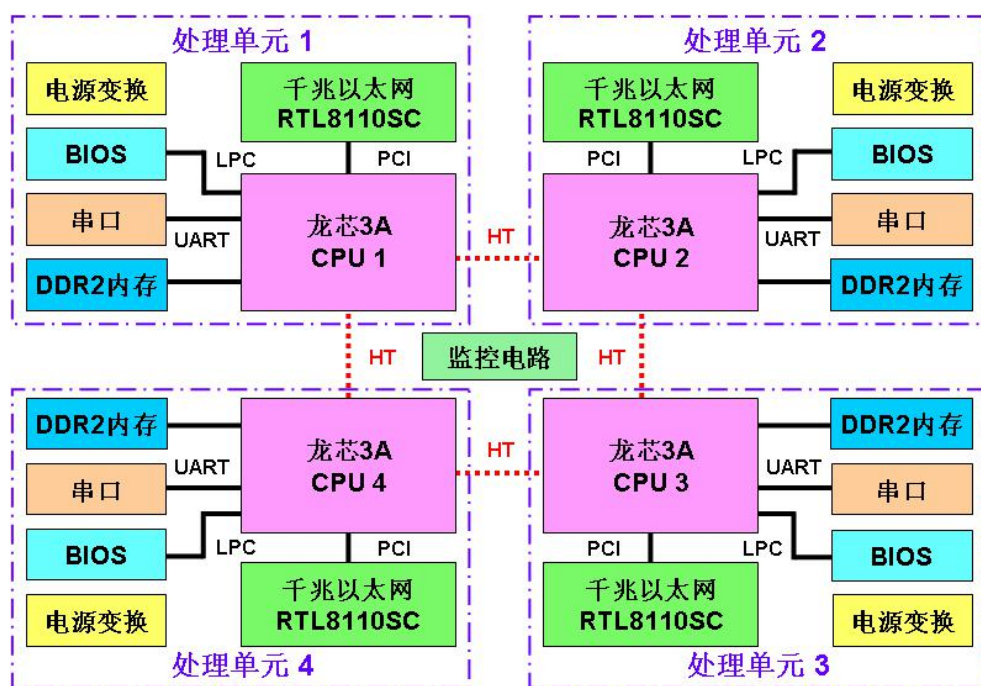


图 1-3 四路龙芯 3A 教学实验箱处理板框图

处理单元使用龙芯 3A 四核处理器，龙芯 3A 四核处理器工作在 825 MHz 时钟频率下，处理器单核的计算能力是 3.3 Gflops，单处理器的计算能力是 13.2 Gflops。PCI 接口工作模式为 32 位，时钟频率 33 MHz，I/O 带宽为 132 MB/s。DDR2 内存的工作频率为 200 MHz，带宽可达 3.2 GB/s。

处理板的设计可满足多种并行层次应用上的需求。在 HT 总线不使用时，实现单处理器结构、SMP 结构和 SMP 集群结构等；在处理器支持 Cache 一致性的

HT 通信并使用 HT 总线的情况下，可以实现 4 路 4 核的 CC-NUMA 结构和 CC-NUMA 集群结构。因此，利用该处理板可以构建实验平台，进行计算机体系结构教学在多种并行层次上的教学实验。

## 1.2. 实验平台运行流程

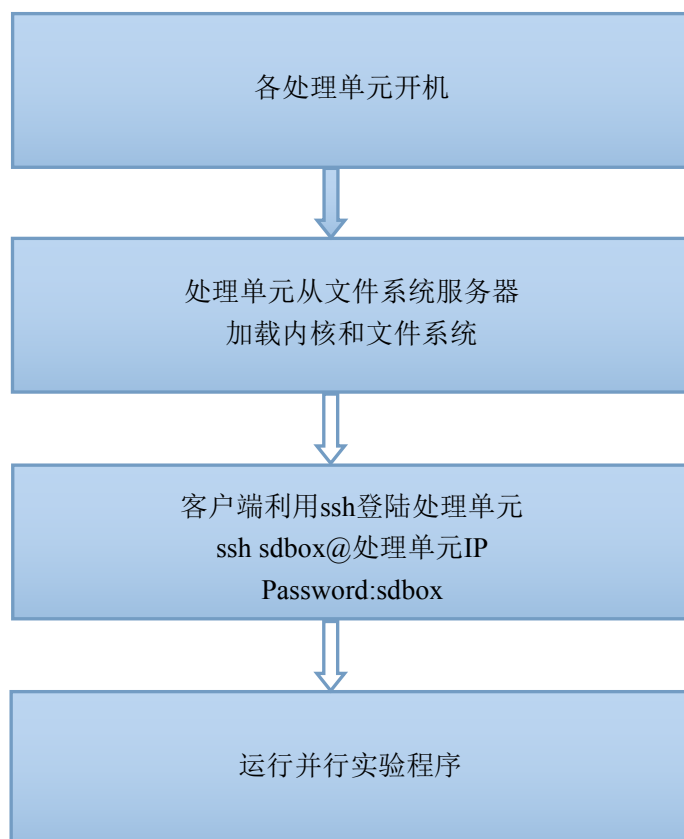


图 1-4 实验平台运行流程

## 2 操作系统安装

### 2.1. 配置文件系统服务器

#### (1) 安装Debian系统

- 系统安装完毕后建议修改/etc/apt/sources.list文件，添加源，如下图所示：

```
#deb http://debian.printk.org/debian squeeze main
#deb http://debian.printk.org/debian unstable main
#deb-src http://debian.printk.org/debian squeeze main

deb http://debian.nctu.edu.tw/debian/ squeeze main non-free contrib
deb http://debian.nctu.edu.tw/debian/ squeeze-proposed-updates main non-free contrib

deb http://mirrors.163.com/debian/ squeeze main non-free contrib
deb http://mirrors.163.com/debian/ squeeze-proposed-updates main non-free contrib
deb http://mirrors.163.com/debian-security/ squeeze/updates main non-free contrib
```

图2-1 添加源

- 添加完毕使用apt-get update更新源

```
root@Loong:~# apt-get update
获取: 1 http://debian.nctu.edu.tw squeeze Release.gpg [1,672 B]
忽略 http://debian.nctu.edu.tw/debian/ squeeze/contrib Translation-en
忽略 http://debian.nctu.edu.tw/debian/ squeeze/contrib Translation-zh
忽略 http://debian.nctu.edu.tw/debian/ squeeze/contrib Translation-zh_CN
忽略 http://debian.nctu.edu.tw/debian/ squeeze/main Translation-en
获取: 2 http://debian.nctu.edu.tw/debian/ squeeze/main Translation-zh [1,685 B]
获取: 3 http://debian.nctu.edu.tw/debian/ squeeze/main Translation-zh_CN [67.3 kB]
忽略 http://debian.nctu.edu.tw/debian/ squeeze/non-free Translation-en
忽略 http://debian.nctu.edu.tw/debian/ squeeze/non-free Translation-zh
忽略 http://debian.nctu.edu.tw/debian/ squeeze/non-free Translation-zh_CN
获取: 4 http://debian.nctu.edu.tw squeeze-proposed-updates Release.gpg [1,571 B]
获取: 5 http://debian.nctu.edu.tw/debian/ squeeze-proposed-updates/contrib Translation-en [14 B]
忽略 http://debian.nctu.edu.tw/debian/ squeeze-proposed-updates/contrib Translation-zh
忽略 http://debian.nctu.edu.tw/debian/ squeeze-proposed-updates/contrib Translation-zh_CN
获取: 6 http://debian.nctu.edu.tw/debian/ squeeze-proposed-updates/main Translation-en [89.4 kB]
78% [6 Translation-en 54.6 kB/89.4 kB 61%] [正在等待报头]
```

图2-2 更新源

#### (2) IP地址配置

- 文件系统服务器默认使用192.168.100.250地址。
- 计算节点IP分配，同文件系统服务器为一个封闭的系统，需使用192.168.100 ip网段进行配置：192.168.100.1-192.168.100.4 。
- 如果想让文件系统服务器能够和外网相连，可采用下述两个配置方案：

1. 服务器安装两个网卡，其中一个网卡配置192.168.100.250
2. 只有一个网卡配置备用ip，命令如下：

```
ifconfig eth1 192.168.100.250
```

#### (3) 配置NFS

文件系统服务器必须支持网络文件系统（NFS），配置成NFS服务器。

- 安装三个软件包，命令如下：



## apt-get install nfs-common nfs-kernel-server portmap

```

root@Loong:~# apt-get install nfs-common nfs-kernel-server portmap
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会安装下列额外的软件包:
  libgssglue1 libnfsidmap2 librpcsecgss3
下列【新】软件包将被安装:
  libgssglue1 libnfsidmap2 librpcsecgss3 nfs-common nfs-kernel-server portmap
升级了 0 个软件包，新安装了 6 个软件包，要卸载 0 个软件包，有 254 个软件包未被升级。
需要下载 540 kB 的软件包。
解压缩后会消耗掉 1,700 kB 的额外空间。
您希望继续执行吗？[Y/n]Y
获取：1 http://debian.nctu.edu.tw/debian/ squeeze/main libgssglue1 mipsel 0.1-4 [22.5 kB]
获取：2 http://debian.nctu.edu.tw/debian/ squeeze/main libnfsidmap2 mipsel 0.23-2 [29.3 kB]
获取：3 http://debian.nctu.edu.tw/debian/ squeeze/main librpcsecgss3 mipsel 0.19-2 [33.4 kB]
获取：4 http://debian.nctu.edu.tw/debian/ squeeze/main portmap mipsel 6.0.0-2 [38.1 kB]

```

图2-3 NFS软件包安装

### • 服务器端配置

#### 1. 创建共享目录

- 建立 /var/lib/diskless/default

```
mkdir /var/lib/diskless/default -p
```

- 建立软连接/srv/nfsroot，指向/var/lib/diskless/default

```
ln -s /var/lib/diskless/default /srv/nfsroot
```

- 把提供的根文件系统解压到/var/lib/diskless/default

```

root@Loong:~# ll -h /srv/
总用量 0
lrwxrwxrwx 1 root root 26  9月 12 16:29 nfsroot -> /var/lib/diskless/default/
root@Loong:~#

```

图2-4 软连接的建立

#### 2. 配置/etc/exports文件

- 配置NFS服务
- 修改/etc/exports文件，添加如下语句：

```

/srv/nfsroot          192.168.100.0/255.255.255.0(rw,no_root_squash,async)
/var/lib/diskless/default 192.168.100.0/255.255.255.0(rw,no_root_squash,async)
/home 192.168.100.0/255.255.255.0(rw,no_root_squash,async)

```



### 3. 重启NFS服务

输入重启指令：`/etc/init.d/nfs-kernel-server restart`

```
root@Loong:~# /etc/init.d/nfs-kernel-server restart
Stopping NFS kernel daemon: mountd nfsd.
Unexporting directories for NFS kernel daemon....
FATAL: Could not load /lib/modules/2.6.36.3+/modules.dep: No such file or
Exporting directories for NFS kernel daemon...exportfs: /etc/exports [2]:
.255.255.0:/srv/nfsroot".
    Assuming default behaviour ('no_subtree_check').
    NOTE: this default has changed since nfs-utils version 1.0.x

exportfs: /etc/exports [3]: Neither 'subtree_check' or 'no_subtree_check'
    Assuming default behaviour ('no_subtree_check').
    NOTE: this default has changed since nfs-utils version 1.0.x

exportfs: duplicated export entries:
exportfs:      192.168.60.0/255.255.255.0:/var/lib/diskless/default
exportfs:      192.168.60.0/255.255.255.0:/var/lib/diskless/default
exportfs: /etc/exports [4]: Neither 'subtree_check' or 'no_subtree_check'
    Assuming default behaviour ('no_subtree_check').
    NOTE: this default has changed since nfs-utils version 1.0.x

Starting NFS kernel daemon: nfsd mountd.
```

图2-6 NFS服务重启

- NFS的启动和停止指令

启动服务：`/etc/init.d/nfs-kernel-server start`

停止服务：`/etc/init.d/nfs-kernel-server stop`

#### (4) TFTP配置

计算节点的linux内核需要从文件系统服务器tftp下载，故需配置TFTP

- 下载地址：`tftp://192.168.100.1/vmlinux`
- 安装tftp

`apt-get install tftpd tftp openbsd-inetd`

```
root@Loong:~# apt-get install tftpd tftp openbsd-inetd
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列【新】软件包将被安装：
  openbsd-inetd tftp tftpd
升级了 0 个软件包，新安装了 3 个软件包，要卸载 0 个软件包，有 254 个软件包未被升级。
需要下载 72.7 kB 的软件包。
解压缩后会消耗掉 311 kB 的额外空间。
获取：1 http://debian.nctu.edu.tw/debian/ squeeze/main openbsd-inetd mipsel 0.20080125-6 [36.0
获取：2 http://debian.nctu.edu.tw/debian/ squeeze/main tftp mipsel 0.17-18 [19.4 kB]
获取：3 http://debian.nctu.edu.tw/debian/ squeeze/main tftpd mipsel 0.17-18 [17.3 kB]
下载 72.7 kB，耗时 4秒 (15.3 kB/s)
```

图2-7 tftp软件包安装

- 配置文件/etc/inetd.conf

该配置文件使用默认即可，其中/etc/inetd.conf 内容为：

`tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /srv/tftp`

- 修改权限

– `sudo chmod -R 777 /srv/tftp`

```
root@Loong:~# mkdir /srv/tftp -p
root@Loong:~# chmod -R 777 /srv/tftp/
root@Loong:~# ls /srv/
nfsroot tftp
root@Loong:~#
```

图2-8 tftp文件夹

- 将结点使用的linux内核命名为vmlinux-smp-4core，并拷贝至/srv/tftp
- 重启inetd服务

`/etc/init.d/openbsd-inetd restart`

```
root@Loong:~# /etc/init.d/openbsd-inetd restart
Restarting internet superserver: inetd.
root@Loong:~#
```

图2-9 重启tftp服务

- 测试tftp

1. 在 192.168.100.250 的 tftp 文件夹里面建立一个文件

`echo "hello world" > hello.txt`

```
root@Loong:/srv/tftp# echo "hello world" > hello.txt
root@Loong:/srv/tftp# cat hello.txt
hello world
root@Loong:/srv/tftp#
```

图 2-10 tftp 测试文件系统服务器端

3. 在可以连接到该主机的计算机上使用 tftp 协议连接

`tftp 192.168.100.1`

`tftp> get hello.txt`

`tftp> quit`

`cat hello.txt`

```
tftp> get hello.txt
Received 13 bytes in 0.0 seconds
tftp> quit
$ cat hello.txt
hello world
```

图2-11 tftp测试客户端

### (5) 配置节点私有的文件系统

- 拷贝192.168.100.1文件系统到/var/lib/diskless/default
- 在 /var/lib/diskless/default 目录运行运行脚本copyfs.sh将在当前目录产生所有70个节点的私有目录，可以根据节点的ip选择相应的目录，删除不必要的目录

## (6) 登录处理单元

从文件系统服务器登陆各处理单元

– ssh sdbox@192.168.100.1

– Password: \*\*\*\*

```
root@Loong:/srv/tftp# ssh 192.168.60.10
The authenticity of host '192.168.60.10 (192.168.60.10)' can't be established.
RSA key fingerprint is 20:65:6e:27:9a:67:e5:c3:83:05:2a:b5:28:ce:be:ce.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.60.10' (RSA) to the list of known hosts.
root@192.168.60.10's password:
Linux debian 2.6.32-5-686 #1 SMP Fri May 10 08:33:48 UTC 2013 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Sep 13 09:17:05 2013 from linux-27.local
```

图2-12 登陆子节点

## 2.2. 启动处理单元

首次启动处理单元需要进行一下步操作。实验箱上电后，按照操作说明启动4个处理器。通过处理器串口分别配置每个处理单元，步骤如下：

### (1) 配置IP

配置网卡的ip地址，并测试网卡是否工作，利用ping命令实现。若文件系统服务器IP为192.168.100.1，处理单元同文件系统服务器为一个封闭的系统，故需使用192.168.100 ip网段进行配置。

命令格式：ifconfig eth0 192.168.100.x

### (2) 加载内核

命令格式：load tftp://192.168.100.1/vmlinux-smp-4core

该地址为文件系统服务器中内核存放地址。

### (3) 启动并加载根文件系统

命令格式：

```
g console=ttyS0,115200 init=/bin/sh rw root=/dev/nfs nfsroot=192.168.100.250:/
home/nfs-debian6 ip=192.168.100.1:192.168.100.250::255.255.0.0:node1:eth0:off
```

• nfsroot=192.168.100.250:/srv/nfsroot 表示网络文件系统服务器的IP地址以及网络文件系统在服务器上的存放路径，见文件系统服务器配置。

- ip=192.168.100.1:192.168.100.250::255.255.0.0

第一个IP地址192.168.100.1为本处理单元IP地址，可随意配置；

第二个IP地址192.168.100.250为文件系统服务器的IP地址；

node1 表示系统启动完毕后登陆的用户名为 node1；

完成以上三步，处理单元系统启动成功，进入linux系统。在以后的使用中，若文件系统服务器配置未改变，处理单元会自动加载内核和文件系统，无须重复设置。

#### (4) 登录处理单元

从文件系统服务器或其他终端远程登陆各处理单元，指令格式：

`ssh sdbox@192.168.100.x Password: ****`

- 192.168.100.x为处理单元IP

## 第二部分 教学实验指导

课程名称:	计算机系统结构实验		
英文名称:			
设置形式:	独立设课	课程模块:	专业核心课
实验课性质:	专业实验	课程编号:	
一、学时、学分			
课程总学时:	实验学时:	课程学分:	

### 1. 教学实验要求及内容

#### 一、适用专业及年级

高等学校计算机及相关专业的本科高年级学生和研究生。

#### 二、课程目标与基本要求

通过本课程的学习,帮助学生在学习计算机系统结构中对基本概念的理解。具体包含定量原理、指令流水、cache行为、主存行为以及多处理机和多计算机结构上的可观测软件行为的实践操作。除了加深对体系结构概念的理解、让学生体验计算机系统结构如何影响软件程序运行之外,也要求学生初步掌握一些充分利用体系结构的软件编程优化方法。

#### 三、主要仪器设备

龙芯4核并行实验平台。

#### 四、实验项目

序号	实验项目名称	实验基本方法和内容	项目学时	项目类型	每组人数	教学要求
1	Amdahl定量原理	编写程序体现amdahl定理				
2	指令流水	编写程序体验数据相关等对流水的影响				
3	LLC队整体效率的影响	观察cache启用和关闭时的访存性能差异				

4	cache乒乓效应实验	观察程序中的伪共享现象				
5	访存行为对cache性能的影响	编写程序测试各级cache容量				
6	cache层次机容量评估	通过软件探测cache层次及其容量				
7	TLB命中及缺失对访存性能的影响	编写程序观察TLB缺失对性能的影响				
8	内存控制器带宽与竞争	编写程序观察贮存带宽竞争情况				
9	访存中的NUMA因素	编写程序测试本地和远程访问性能差异				
10	多和结构体验——OpenMP	编写OpenMP程序观察多核并行计算				
11	集群结构体验——MPI	编写MPI程序观察多计算机并行计算				

## 五、考核方式及成绩评定

### ➤ 考核方式

1、实验平时成绩：（1）实验考勤：每次考勤分出勤（2分）；请假、迟到、早退（1分）；旷课（0分）记分。（2）预习报告：要求写明实验目的、主要实验设备名称、实验原理和内容。分优秀（4分）、良好（3分）、中等（2分）、及格（1分）和不及格（0分）记分。（3）实验报告：要求写明实验设备名称和型号、实验步骤、实验分析及注意事项。分优秀（4分）、良好（3分）、中等（2分）、及格（1分）和不及格（0分）记分。2、实验考试：上机考试，满分为100分。

### ➤ 成绩评定

总实验成绩占本课程成绩的20%。 总实验成绩 = 实验平时成绩×50%+实验考试成绩×50%。

## 六、实验教科书、参考书

1. 实验教科书
2. 实验参考书



## 第三部分 实验操作指导

### 实验 1 Amdahl 定量原理

#### 1、实验目的

验证Amdahl定理, 进而加深对Amdahl定理的理解。

#### 2、实验原理

并行计算中的加速比是用并行前的执行速度和并行后的执行速度之比来表示的, 它表示了在并行化之后的效率提升情况。加速比 (SpeedUP) 可用公式:

$$S = \frac{W_s + W_p}{W_s + \frac{W_p}{P}}$$

来表示。式中  $W_s, W_p$  分别表示问题规模的串行分量 (问题中不能并行化的那一部分) 和并行分量,  $P$  表示处理器数量。

只要注意到当  $P \rightarrow \infty$  时, 上式的极限是  $\frac{W}{W_s}$ , 其中,  $W = W_s + W_p$ 。这意味着无论我们如何增大处理器数目, 加速比是无法高于这个数的。

Amdahl (阿姆达尔) 定理是固定负载 (计算总量不变时) 时的加速比量化标准。如果,  $f$  表示计算机执行某个任务的总时间可被改进部分的执行时间所占百分比, 则  $W_s + W_p$  可以相应地表示为  $f + (1-f)$ , 上述公式改为:

$$S = \frac{f + (1-f)}{f + \frac{1-f}{P}} = \frac{P}{1 + f(P-1)}$$

当  $P \rightarrow \infty$  时, 极限为  $1/f$ 。

#### 3、实验内容

实验思想为: 固定运算总量和程序中串行任务和并行任务的比例, 通过不断提高处理器的数量, 并计算出相应的加速比, 观测最终的加速比是否趋向于理论加速比的极限。

样例代码使用PI值计算的OpenMP程序, 编程中固定计算Pi的迭代次数 `NUM_ITERS`, 再把迭代过程按比例  $p$  拆分成两部分 ( $0$  至  $LOOP1$  和  $LOOP1$  至  $LOOP2$ ),

前半部分使用串行计算，后半部分使用openmp进行并行计算。

(1)把任务全部分给串行代码，运行得到串行运行时间 $T_s$ 。

(2)把任务按1:4的串/并比例划分，再分别用不同的处理器数运行，得到 $T_p$ ，除以 $T_s$ 得到相应的加速比。

(3)尝试按其他比例划分任务，再重复(2)，并记录运行时间。

### 实验样例代码：

```
/*
 * Calculate  $\pi = 4 \int_0^1 \sqrt{1 - x^2} \, dx$ 
 * using a rectangle rule  $\sum_{i=1}^N f(x_0 + i h - h/2) h$ 
 *
 * Compile as  $\$> gcc -Wall -Wextra -fopenmp -lm$ 
 * Run as  $\$> OMP_NUM_THREADS=2 PERCENTAGE=30 ./a.out$ 
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

const long long NUM_ITERS = 100000000;
int percentage = 30;
long long LOOP1;
long long LOOP2;

void init_percentage()
{
    char *s = getenv("PERCENTAGE");
    if (s)
        sscanf(s, "%d", &percentage);

    LOOP1 = NUM_ITERS / 100. * percentage;
    LOOP2 = NUM_ITERS;
}

int main(int argc, char *argv[])
{
    const double L = 1.0;
    const double h = L / NUM_ITERS;
    const double x_0 = 0.0;

    double pi;
    double s_t, p_t, t_1, t_2;
```

```
double total_time = 0;

int i;
double sum = 0.0;

init_percentage();

/* serial */
t_1 = omp_get_wtime();

for (i = 0; i < LOOP1; ++i)
{
    double x = x_0 + i * h + h/2;
    sum += sqrt(1 - x*x);
}

t_2 = omp_get_wtime();
s_t = t_2 - t_1;

/* parallel */

t_1 = omp_get_wtime();

#pragma omp parallel for reduction(+: sum) schedule(static)
for (i = LOOP1; i < LOOP2; ++i)
{
    double x = x_0 + i * h + h/2;
    sum += sqrt(1 - x*x);
}

t_2 = omp_get_wtime();
p_t += t_2 - t_1;
total_time += s_t + p_t;
pi = sum * h * 4.0;

if (argc == 1) {
    printf("pi ~ %f\n", pi);
    printf("serial time: %f\n", s_t);
    printf("parallel time: %f\n", p_t);
    printf("omp_get_max_threads(): %d\n", omp_get_max_threads());
    printf("Total time: %f\n", total_time);
} else {
    printf("%d\t%d\t%f\t%f\t%f\n",
```

```

        percentage,
        omp_get_max_threads(),
        s_t,
        p_t,
        total_time);
    }

    return 0;
}

```

### 编译和运行的方法：

编译 openmp 程序的时候需要加 *-fopenmp* 参数，同时因为程序使用了数学库，所以命令中需要加 *-lm* 参数。另外，运行程序的时候，我们通过 *OMP\_NUM\_THREADS* 和 *PERCENTAGE* 环境变量分别控制运行的并行计算的线程数和串/并任务划分百分比。

```

sdbox@loongsonbox-n1:~$ gcc -lm -fopenmp Amdahl.c -o amdahl
sdbox@loongsonbox-n1:~$ OMP_NUM_THREADS=4 PERCENTAGE=25 ./amdahl
pi ~ 3.141593
serial time: 10.567035
parallel time: 7.905465
omp_get_max_threads(): 4
Total time: 18.472500
sdbox@loongsonbox-n1:~$ OMP_NUM_THREADS=8 PERCENTAGE=25 ./amdahl
pi ~ 3.141593
serial time: 10.566905
parallel time: 3.974850
omp_get_max_threads(): 8
Total time: 14.541755
sdbox@loongsonbox-n1:~$ OMP_NUM_THREADS=16 PERCENTAGE=25 ./amdahl
pi ~ 3.141593
serial time: 10.567010
parallel time: 2.000749
omp_get_max_threads(): 16
Total time: 12.567759
sdbox@loongsonbox-n1:~$

```

### 实验数据：

表1-1 不同处理器和任务串并比下的Pi值计算时间

percent	threads	serial_time	parallel_time	total_time	speedup
25	1	11.506036	34.328689	45.834725	1.004185
	2	11.505946	17.175143	28.681089	1.60477
	4	11.521223	8.597357	20.11858	2.287764
	6	11.505748	5.745975	17.251723	2.66794
	8	11.506175	4.342707	15.848883	2.904089
	12	11.509832	2.909625	14.419456	3.191976
	16	11.505042	2.240095	13.745137	3.348571
50	1	23.028475	22.877514	45.905989	1.002627
	2	23.011621	11.448022	34.459644	1.335666

	4	23.015012	5.73424	28.749252	1.600966
	6	23.011336	3.840856	26.852193	1.714071
	8	23.034833	2.905104	25.939937	1.774351
	12	23.031993	1.975208	25.007201	1.840532
	16	23.026538	1.520482	24.54702	1.875037
75	1	34.505526	11.438027	45.943553	1.001807
	2	34.534692	5.722439	40.25713	1.143315
	4	34.520478	2.879258	37.399735	1.230666
	6	34.519439	1.936925	36.456364	1.262511
	8	34.520154	1.461912	35.982066	1.279153
	12	34.518964	1.014053	35.533017	1.295318
	16	34.520181	0.821263	35.341444	1.30234
100	1	46.026248	0.000316	46.026564	1

percent 串行任务占并行任务的百分比  
 threads 线程数  
 serial\_time 串行任务的运行时间  
 parallel\_time 并行任务的运行时间  
 total\_time 任务运行的总时间  
 speedup 加速比

## 5、实验分析

将表 1-1 的数据绘制图表如图 1-1 所示。

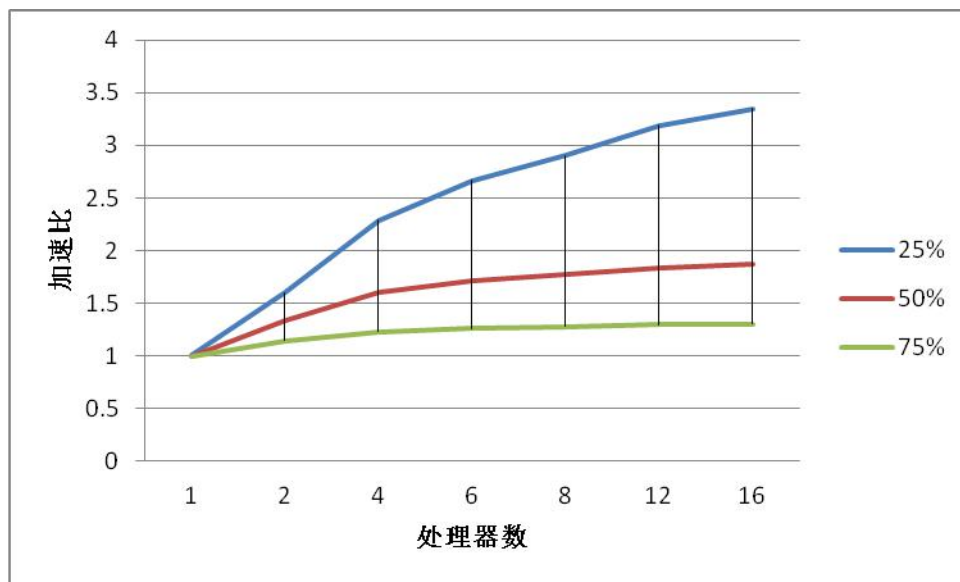


图 1-1 加速比与处理器数量、任务串并比的关系

通过观察上图，我们可以发现，随着处理器的增加，实验的到的加速比也随之增加，但是加速越来越不明显。上述曲线上升趋势与  $(1/f)$  什么关系？这与 Amdahl 定理的描述相一致吗？

## 实验 2 循环展开与指令流水

(2F 处理器适用, 实验箱 3A 处理器需作相应的调整)

### 1 实验目的

- (1) 加深对指令流理解。
- (2) 掌握利用流水功能的编程技巧。

### 2 实验原理

龙芯 3A 采用先进的四发射超标量超流水结构, 采用简单指令集, 类似于 MIPS 指令集。每个处理器核具有如下特点: 支持 MIPS64 指令集及龙芯扩展指令集; 9 级超流水线结构; 四发射乱序执行结构; 2 个定点单元、2 个浮点单元和 1 个访存单元;

为提高处理器执行指令的效率, 把一条指令的操作分成多个细小的步骤, 每个步骤由专门的电路完成。龙芯 3A 的基本流水线包括取指、预译码、译码、寄存器重命名、调度、发射、读寄存器、执行、提交等 9 级。每个阶段都要花费一个时钟周期, 如果没有采用流水线技术, 那么这条指令执行需要 9 个时钟周期; 如果采用了指令流水线技术, 那么当这条指令完成“取指”后进入“预译码”的同时, 下一条指令就可以进行“取指”了, 这样就提高了指令的执行效率。

当指令之间不存在相关时, 它们在流水线中是可以重叠起来并行执行的。这种指令序列中存在的潜在并行性称为指令级并行 (Instruction-Level parallelism, ILP)。流水线处理器的实际 CPI:

$$\text{CPI 流水线} = \text{CPI 理想} + \text{各类停顿周期数的总和}$$

流水线的理想 CPI 是流水线的最大流量。各类停顿包括:

- (1) 结构相关停顿: 是由于两条指令使用同一个功能部件而导致的停顿。
- (2) 控制相关停顿: 是由于指令流的改变 (如分支指令) 而导致的停顿。
- (3) RAW、WAR 和 WAW 停顿: 由数据相关造成的。

减少其中的任何一种停顿, 都可以有效地减少 CPI, 从而提高流水线的性能。

循环级并行: 使一个循环中的不同循环体并行执行。开发循环体中存在的并行性最常见、最基本是指令级并行研究的重点之一例如, 考虑下述语句:



```
for (i=1; i<=500; i=i+1)
```

```
    a[i]=a[i]+s;
```

每一次循环都可以与其他的循环重叠并行执行；在每一次循环的内部，却没有任何的并行性。

### 3 实验内容

本实验采用循环级并行，即使一个循环中的不同循环体并行执行。用一加法程序对比无循环级并行与有循环级并行的运行时间。

程序示例：

```

                for (i=1; i<=200000; i++)
                    x[i] = x[i] + s;
    其中：
        X[]为浮点数组
        S 为一浮点数

```

无循环展开 MIPS32 指令代码：

```

Loop: LWC1    $F0,0 (%0)    // 取一个向量元素放入$F0
      ADD.S   $F0,$F0,$F2    // 加上在 F2 中的常量
      SWC1    $F0,0 (%0)    // 保存结果
      ADDIU   %0,4          // 指针加 4（每个数据占 4 个字节）
      ADDIU   %2,-1         // 循环数减 1
      BNE     %2, $0,Loop    // 如果$2 不等于 0，未结束，继续
    其中：
    整数寄存器 %0：指向向量中的当前元素，初值为向量中最低端元素的地址）
    整数寄存器 %2：循环数。
    浮点寄存器 $F2：用于保存常数 temp。

```

有循环展开 MIPS32 指令代码：

```

"Loop:\n"
"lwc1$f0,0(%0)\n"
"lwc1$f6,4(%0)\n"
"lwc1$f10,8(%0)\n"
"lwc1$f14,12(%0)\n"
"add.s$f4,$f0,$f2\n"
"add.s$f8,$f6,$f2\n"
"add.s$f12,$f10,$f2\n"
"add.s$f16,$f14,$f2\n"
"swc1$f4,0(%0)\n"
"swc1$f8,4(%0)\n"
"swc1$f12,8(%0)\n"
"swc1$f16,12(%0)\n"
"addiu%2,-4\n"
"addiu%0,16\n"
"bne%2,$0,Loop\n"

```

} 循环展开四次

## 实验结果

说明：运行时间单位为时钟数

C 程序	无循环级并行	有循环级并行
260, 000	140, 000	100, 000

(1) 由于 C 程序需要编译，相对来说比汇编程序执行慢，且编译后的指令没有 MIPS32 指令执行效率高，所以执行时间最长。

(2) 无循环级并行的程序，是按 9 级流水顺序执行，存在着等空转时钟周期，有循环级并行的汇编程序，由于提高了并行级，每次循环执行时间相对减少，减少循环次数，执行时间最短。

## 4 实验分析

流水线使指令重叠并行执行，可以达到提高性能的目的，如果指令间没有相关性，可以并行执行。本实验通过高级语言与汇编程序、并行与非并行程序运行时间的对比，体现了指令并行的优越性，可以进一步加深对传统体系结构的理解。

实验可以进一步优化，比如由于数据相关性的存在，可以通过指令调度减少

数据相关性，提高执行效率。

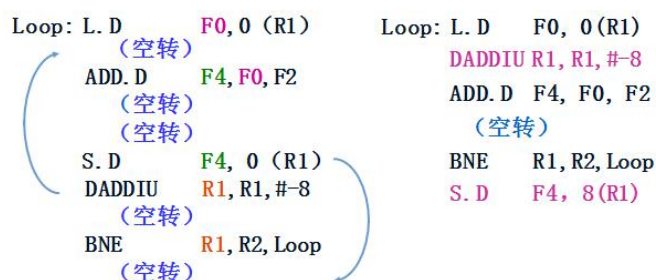


图 2-1 数据相关及其指令顺序调整示意图

如上图所示，指令间存在着很多空转现象，左边程序循环一次需要 10 个时钟周期，其中 5 个都是空转，而进行指令调度后，右边程序循环一次需要 6 个时钟周期，只有一个为空转，很大程度上提高了执行效率。

由此可知，实验可以将循环展开与指令调度结合在一起，更好的体现指令并行，进而减少执行时间，充分体现 CPU 内核中并发计算的效果。

## 实验 3 LLC 对整体效率的影响

(2F 处理器适用，实验箱 3A 处理器需作相应的调整)

### 1 实验目的

- (1) 体验处理器 LLC (Last Level Cache) 对程序对程序整体性能的影响。
- (2) 掌握 Linux 中启用和关闭龙芯处理器 LLC 的方法。

### 2 实验原理

Linux 中开启和关闭 LLC 需要修改内核代码，因此就要跟踪内核的启动过程找到 cache 初始化的地方进行修改，对学生而言具有一定挑战性。但由于我们仅作小修改，相关修改内核的工作不是想象中的那么困难。

首先，下载 linux 内核，本实验使用的是 linux-2.6.27.6 和 linux-2.6.32.2 这两个版本内核，但是编译内核时使用的是 linux-2.6.27.6。

然后，解压源码，可以利用 linux 环境解压，更直接的方法就是直接用 windows 下的 winrar 或者 7z 打开；打开 Source Insight，创建一个工程，然后找到解压后的源码所在的位置，这样就可以在工程中查看源码了(如图 3-1)。

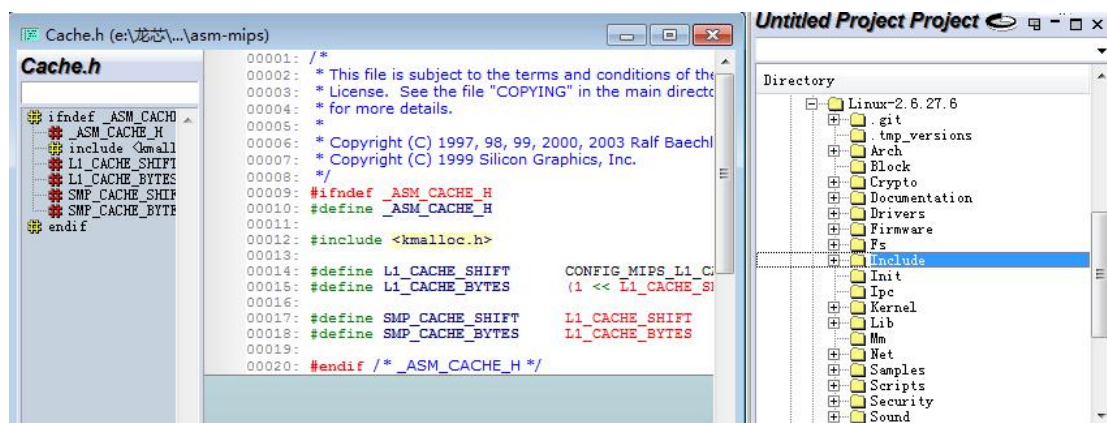


图 3-1 Source Insight 工程缩略图

接着就是查找和定位相关的源码，着手修改内核。龙芯是基于 mips 体系结构的微处理器，而且把 bios 和 bootloader 做成了 pmon，这样就是利用 pmon 来初始化 RAM、加载内核（具体的内核启动顺序，可以参考《UNIX 操作系统设计》、《MINIX 操作系统设计与实现》或者网络等）。这样我们就可以找到 /include/asm-mips/cache.h、cachectl.h、cacheops.h 这几个是实现 cache 的一些初始化定义，/include/asm-mips/cpu-info.h、cpu.h、cpu-feature.h 中是对 cpu 的初始化定

义，也可以找到 TLB 的初始化。

通过查看 cpu.h 文件,看到 cpu\_type\_enum 所支持的 cpu 列表,可以看到 MIPS64 class processors 中支持 CPU-LOONGSON2(如图 4.2), 这样就可以确定 arch 中有相关的配置。

```
/*
 * MIPS64 class processors
 */
CPU_5KC, CPU_20KC, CPU_25KF, CPU_SB1, CPU_SB1A, CPU_LOONGSON2,
CPU_LAST
```

图 3-2 MIPS64 class processors

在 cpu-info.h 中可以看到 cache 的相关结构配置,比如路数、每路的大小、cache line 的大小等等(如图 3-3), 定义二级 cache 的内容(如图 3-4)。

```
/*
 * Descriptor for a cache
 */
struct cache_desc {
    unsigned int waysize; /* Bytes per way */
    unsigned short sets; /* Number of lines per set */
    unsigned char ways; /* Number of ways */
    unsigned char linesz; /* Size of line in bytes */
    unsigned char waybit; /* Bits to select in a cache set */
    unsigned char flags; /* Flags describing cache properties */
};
```

图 3-3 cache\_desc 结构体

```
struct cpuinfo_mips {
    unsigned long    udelay_val;
    unsigned long    asid_cache;

    /*
     * Capability and feature descriptor structure for MIPS CPU
     */
    unsigned long    options;
    unsigned long    ases;
    unsigned int     processor_id;
    unsigned int     fpu_id;
    unsigned int     cputype;
    int              isa_level;
    int              tlbsize;
    struct cache_desc icache; /* Primary I-cache */
    struct cache_desc dcache; /* Primary D or combined I
    struct cache_desc scache; /* Secondary cache */
    struct cache_desc tcache; /* Tertiary/split secondary
    int              srsets; /* Shadow register sets */
    int              core; /* physical core number */
```

图 3-4 cpuinfo\_mips 中的定义

在 cpu\_feature.h 中可以看到 CP0 协处理器的操作,通过手册可以知道 config 寄存器可以实现控制 cpu 的状态和报告 cpu 的当前状态。具体 CP0 寄存器(如图 3-6), 可以知道 CP0 寄存器 config/config1 可以对 cache 的设置。config 寄存器的相关介绍(如图 3-5), 具体域的描述参见龙芯用户手册。

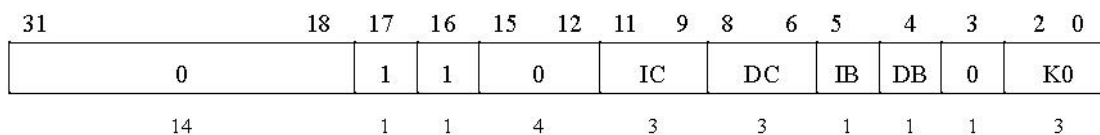


图 3-5 config 寄存器的相关介绍

寄存器号	寄存器名字	描述
0	Index	可写的寄存器，用于指定需要读/写的 TLB 表项
1	Random	用于 TLB 替换的伪随机计数器
2	EntryLo0	TLB 表项低半部分中对应于偶虚页的内容(主要是物理页号)
3	EntryLo1	TLB 表项低半部分中对应于奇虚页的内容(主要是物理页号)
4	Context	32 位寻址模式下指向内核的虚拟页转换表 (PTE)
5	Page Mask	设置 TLB 页大小的掩码值
6	Wired	固定连线的 TLB 表项数目 (指不用于随机替换的低端 TLB 表项)
7		保留
8	BadVaddr	错误的虚地址
9	Count	计数器
10	EntryHi	TLB 表项的高半部分内容 (虚页号和 ASID)
11	Compare	计数器比较
12	Status	处理器状态寄存器
13	Cause	最近一次例外的原因
14	EPC	例外程序计数器
15	PRID	处理器修订版本标识号
16	Config	配置寄存器 (Cache 大小等)
17	LLAddr	链接读内存地址
18	Watch	虚地址空间访问陷阱地址
19		保留
20	Xcontext	64 位寻址模式下指向内核的虚拟页转换表 (PTE)
21		保留



22	Diagnose	使能/禁用 BTB,RAS 以及清空 ITLB 表
23		保留
24	PCLo	性能计数器的低半部分
25	PCHi	性能计数器的高半部分
26		保留
27		保留
28	TagLo	CACHE TAG 寄存器的低半部分
29	TagHi	CACHE TAG 寄存器的高半部分
30	ErrorEPC	错误例外程序计数器
31		保留

图 4.6 CP0 寄存器描述

了解了 CP0 寄存器特别是和 cache 相关的 config 寄存器的初步一些描述后，可以通过内核中 cpu 参数的初始化函数实现 cache 特别是二级 cache 的参数设置。查看/arch/mips/kernel/cpu-probe.c, 此文件介绍了 cpu 的参数配置以及配置函数等，通过查找 cpu\_probe\_legacy 函数，知道完成 loongson2 的初始化的一些函数（如图 3-7），R4K\_OPTS 等，转到 R4K\_OPTS 的定义#define R4K\_OPTS (MIPS\_CPU\_TLB | MIPS\_CPU\_4KEX | MIPS\_CPU\_4K\_CACHE | MIPS\_CPU\_COUNTER)，知道 MIPS\_CPU\_TLB 和 MIPS\_CPU\_4K\_CACHE 这两个可以实现 TLB 和 Cache，这样其实基本能确定初始化 cpu Cache 的函数了，为了明确一下，找到/arch/mips/mm/cache.c，这个文件中就包括所有 cache init 的封装，封装的函数为 cpu\_cache\_init，找到此函数 cpu\_has\_4k\_cache，然后在回到 cpu-fearture.h 中找到关于 cpu\_has\_4k\_cache 的定义（如图 3-8），这样就很容易的知道 loongson2cpu 的 cache 操作函数为 cpu\_has\_4k\_cache。接下来就是要找到这个函数的实现代码，在 c-r4k.c 中可以找到此函数，然后在找到 probe\_pcache 函数，这个函数实现的是各个类型 cpu 的 ICache 和 DCache 的初始化，相应的可以看到 probe\_sache，这个是对二级 cache 的初始化，也就是大小、路数等的赋值，保存于结构体中，mips 专门有对龙芯的 sc 的初始化（图 4.9），可以看到相应的结构体中的参数的初始化。这个是在 setup\_scache 函数中调用的。到此，已经可以定位到二级 cache 的源码位置。

```

case PRID_IMP_LOONGSON2:
    c->cputype = CPU_LOONGSON2;
    c->isa_level = MIPS_CPU_ISA_III;
    c->options = R4K_OPTS |
                MIPS_CPU_FPU | MIPS_CPU_LLSC |
                MIPS_CPU_32FPR;
    c->tlbsize = 64;
    break;

```

图 3-7 loongson2 的配置函数

```

#ifndef cpu_has_4k_cache
#define cpu_has_4k_cache (cpu_data[0].options & MIPS_CPU_4K_CACHE)

```

图 3-8 cpu\_has\_4k\_cache 定义

```

#if defined(CONFIG_CPU_LOONGSON2)
static void __init loongson2_sc_init(void)
{
    struct cpuinfo_mips *c = &current_cpu_data;

    scache_size = 0;
    c->scache.linesz = 0;
    c->scache.ways = 0;
    c->scache.waybit = 0;
    c->scache.waysize = scache_size / (c->scache.ways);
    c->scache.sets = scache_size / (c->scache.linesz * c->scache.ways);
    pr_info("Unified secondary cache %ldkB %s, linesize %d bytes.\n",
            scache_size >> 10, way_string[c->scache.ways], c->scache.linesz);

    c->options |= MIPS_CPU_INCLUSIVE_CACHES;
}
#endif

```

图 3-9 loongson 的二级 cache 的初始化

### 3 实验内容

#### 内核代码修改

经过上面的“艰辛万苦”，发现原来涉及二级 cache 的就那么一点，这样就可以修改那些参数了。具体修改可以参照你所做试验的要求，这里我把二级 cache 的相应参数设置为 0，如果其它值，可以参考公式(自己修改)： $\text{waysize} = \text{cache} / \text{ways}$ ， $\text{sets} = \text{cache} / \text{linesz}$ ，也可以根据组相联的介绍来实现。

#### 编译内核

修改完就要编译并运行，这样才可以查看二级 cache 的关闭效果。这时，所用环境：win 7，虚拟机 VMware workstation、虚拟机系统 ubuntu10.04、交叉编译工具 cross-chain。

解压内核 `tar -zxvf linux-2.6.27.6.tar.gz`，可以放到你想要放的地方；修改 Makefile 文件中的 ARCH 和 CROSS\_COMPILE 这两项，分别更改为 ARCH=mips CROSS\_COMPILE=mis64el-linux-（注意：这个就是交叉编译工具，你可以把编译工具所在的目录放在此处，亦可以通过修改 ~/.bashch，在最后添加 `export PATH=$PATH:/opt/cross-chain/bin`，这样就可以添加全局环境），然后：wq 保存退出；

cp arch/mips/config/fuloong\_deconfig .config,这个配置文件，就是你目标机的配置文件，然后 make menuconfig,这个是通过界面进行你想要的内核的 cpu type、filesystem 等等的配置，做完这些就可以进行 make 编译内核了，编译完后会在你解压内核目录下生成 vmlinux 和 vmlinux.32 文件，这两个都可以。这样你就可以把内核复制到目标机的硬盘进行测试了。

### 内核加载以及测试

龙芯的内核加载很简单，开机进入 pmon，然后命令 devls 查看所支持的硬盘类型——sata0，这样就可以加载内核：load /dev/fs/ext2@sata0/boot/vmlinux1（这是刚才编译后拷贝过来的内核文件名），然后 g console=tty root=/dev/sda1。这样内核加载成功，进行启动。因为文件系统已经存在所以进入系统，拷入先前在 vmlinux 内核下的程序运行。

### 测试程序源码：

```
/*
 * cachetest.c
 *
 * Created on: 2012-4-7
 * Author: dragon
 */
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<unistd.h>
#include<sys/types.h>

int main(int argc,char **argv)
{
    int i,j,k=0;
    long t3;
    struct timeval tv1,tv2;
    //t1=time((time_t*)NULL);
    gettimeofday(&tv1,NULL);
    for(i=0;i<10000;i++)
    {
        for(j=0;j<10000;j++)
        {
            k=k+2;
        }
    }
}
```

```
//t2=time((time_t*)NULL);
gettimeofday(&tv2,NULL);
//printf("%d %d\n",tv1.tv_sec,tv1.tv_usec);
//printf("%d %d\n",tv2.tv_sec,tv2.tv_usec);
t3=(long)tv2.tv_sec*1000000+(long)tv2.tv_usec
    -(long)(tv1.tv_sec*1000000+tv1.tv_usec);
printf("运行时间: %ld\n",t3);
return 0;
}
```

## 实验结果

```
sdbox@loongsonbox-n1:~$ gcc cachetest.c -o cachetest
sdbox@loongsonbox-n1:~$ ./cachetest
运行时间: 2570647
sdbox@loongsonbox-n1:~$
```

在原内核下：运行时间：3685472us

在新内核下：运行时间：2570647us

经过对比可以很明显的发现，关闭二级 cache 后，程序所运行的时间接近原来的两倍。虽然此时一级 cache（Icache 和 Dcache）仍在工作，但也明显体现出二级 cache 对程序运行整体性的影响。

## 4 实验分析

通常说 cache 比主存访问速度要快几倍，为何我们的实验中程序执行时间差异不那么明显？如果 1 级 cache 也可以关闭，程序性能下降会更明显吗？

## 实验 4 cache 乒乓效应实验

### 1 实验目的

- (1) 了解 cache 乒乓效应。
- (2) 掌握避免产生 cache 乒乓效应的编程方法。

### 2 实验原理

cache 的乒乓效应是由于 cache 数据假共享所引起。在多处理机系统中，不同的处理器可能需要同一个主存块的不同部分而不是相同的字节。尽管实际数据不共享，但如果一个处理器对该块的其他部分写入，则这个块在其他高速缓存上的拷贝就要全部地进行更新或者使无效。这就是所谓的假共享。它对系统的性能有负面的影响。

如图 4-1 是假共享引起 cache 乒乓效应的示意图。两个处理器 CPU0 和 CPU1 访问同一主存块中的不同部分，当 CPU0 更新了 0 号主存块中的数据，根据高速缓存一致性协议将会更新或使 CPU1 中的高速缓存块无效，而在此时 CPU1 更新了 1 号主存块中的数据，调整缓存一致性协议反过来又会使 CPU0 中的高速缓存块无效，如此反复下去就会导致高速缓存块的乒乓效应。

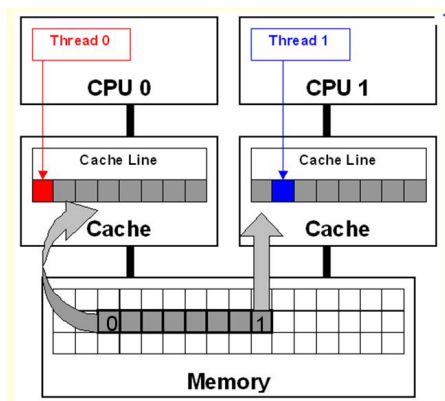


图 4-1 高速缓存假共享示意图

而我国自主研发的龙芯 3 CPU 片内采用二维 mesh 互连结构，其中每个结点由 8\*8 的交叉开关组成，每个交叉开关连接四个处理器以及分成四个体的共享二级 Cache，并与东 (E) 南 (N) 西 (W) 北 (N) 四个方向的其他结点互连。因此，2\*2 的 mesh 可以连接 16 个处理器，4\*4 的 mesh 可以连接 64 个处理器。龙芯 3 CPU 片内二级 Cache 分布在不同的结点中，被片内的所有处理器共享，在二级 Cache

是不存在乒乓效应（仅在 1 级 cache 上有该效应）。

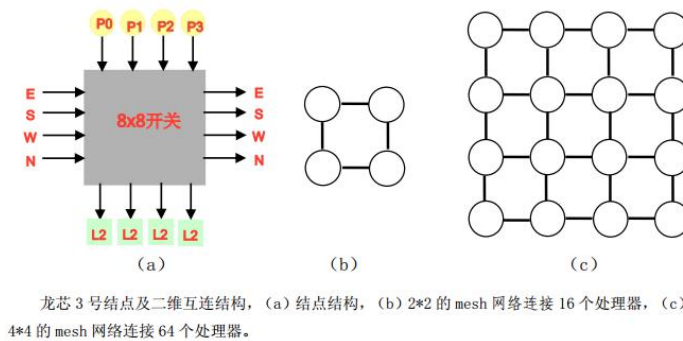


图 4-2 龙芯核间互连

本实验的实验目的就是要验证采用交叉开关的龙芯 3 的 Cache 乒乓效应的具体表现，并对比 X86 CPU 的计算机的乒乓效应对程序性能影响的具体数值。

### 3 实验内容

首先编写出程序 `cache_pingpong`，程序首先新建一个包含 33 个 `int` 元素的数组 `array`，然后创建线程 `threadA` 和 `threadB` 同时访问该数组中的不同的两个元素，对该元素进行累加（加一）操作 `MAXVALUE` 次数。最后统计两个线程访问不同元素所花时间，打印这个两个元素的值以验证线程对元素的正确访问。

第一次我们编写代码，`threadA` 和 `threadB` 两个线程分别对 `array[0]`, `array[1]` 的值进行累加，编译源程序得到可执行文件名为 `has_pingpong` 并运行程序，查看运行结果。

第二次我们修改代码，`threadA` 和 `threadB` 两个线程分别对 `array[0]`, `array[20]` 的值进行累加，编译源程序得到可执行文件名为 `no_pingpong` 并运行程序，查看运行结果。

比较两次运行结果中打印的时间值。

由于 `array[0]` 和 `array[1]` 的地址空间相差 4 字节，访问这两个元素时，他们以很大的概率会被缓冲到同一 cache 行上，不同的线程同时访问这两个元素就会产生乒乓效应。而 `array[0]` 和 `array[20]` 内存地址距离相差 80，如果 cache 行大小小于 80 字节，那么这两个元素就一定不在同一 cache 行上面，访问这两个元素就不可能产生乒乓效应。因此该数组第一个元素和最后一个元素的内存地址距离相差 128，如果 cache 行大小小于 128 字节，那么这两个元素就一定不在同一



cache 行上面，访问这两个元素就不可能产生乒乓效应。

### 样例源代码

这里附上程序 cache\_pingpong 的源程序。

```
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>
#include <unistd.h>
#define MAXVALUE 100000000
int array[33];
void * addx(void *arr)
{
    int i;
    int *array=(int *)arr;
    *array=0;
    for(i = 0;i<MAXVALUE;i++)
    {
        *array+=1;
    }
    return NULL;
}

void * addy(void *arr)
{
    int i;
    int *array=(int *)arr;
    *array=0;
    for(i = 0;i<MAXVALUE;i++)
    {
        *array += 1;
    }
    return NULL;
}

int main()
{
    struct timeval tpStart, tpEnd;
    float timeUse;
    pthread_t threadA, threadB;

    gettimeofday(&tpStart, NULL);
    pthread_create(&threadA, NULL, addx, (void *)&array[0]);
```

```

pthread_create(&threadB, NULL, addy, (void *)&arry[20]);
pthread_join(threadA, NULL);
pthread_join(threadB, NULL);
gettimeofday(&tpEnd, NULL);
timeUse =
1000000*(tpEnd.tv_sec-tpStart.tv_sec)+tpEnd.tv_usec-tpStart.tv_usec;
printf("%ld\n", arry[0]);
printf("%ld\n", arry[20]);
printf("the cost of total time is %f us\n", timeUse);
return 0;
}

```

## 编译和运行

因为使用了pthread线程库, 所以, 我们在编译程序的时候得-lpthread参数

```

sdbox@loongsonbox-n1:~$ ./no_pingpong
100000000
100000000
the cost of total time is 3947761.000000 us
sdbox@loongsonbox-n1:~$ ./has_pingpong
100000000
100000000
the cost of total time is 15375796.000000 us
sdbox@loongsonbox-n1:~$

```

图4-3 编译与运行

## 实验结果

首先在 X86 平台下将程序 (X86 平台上编译的)no\_pingpong 运行 10 次, 得出如下实验结果, 如表 4-1 所示各 time\_A\_X86 值, 并计算机出平时运行时间。

表 4-1 程序 no\_pingpong 运行结果 time\_A\_X86 值

次数	1	2	3	4	5	平均时间 (us)
时间(us)	249820	248977	253827	245156	246354	248923.5
次数	6	7	8	9	10	
时间(us)	254172	249895	243960	249738	247336	

再同样在 X86 平台下将程序 has\_pingpong 运行 10 次, 得出如下实验结果, 分别如表 4-2, 并计算机出平时运行时间。

表 4-2 程序 has\_pingpong 运行结果 time\_B\_X86 值

次数	1	2	3	4	5	平均时间(us)
时间(us)	660638	652247	1088059	952505	920920	800198.6

次数	6	7	8	9	10	
时间(us)	656289	1099344	650376	671703	649905	

从上表中各程序的平均运行时间可看出, has\_pingpong 的平均运行时间明显高于 no\_pingpong, 这正是由于程序中各线程在运行过程中数据假共享产生 cache 乒乓效应造成的。

在龙芯 3A 平台下将程序（龙芯 3A 平台下编译的）no\_pingpong 和 has\_pingpong 各运行 10 次, 得出如下实验结果, 如表 4-3 和表 4-4 所示, 并计算机出平时运行时间。

表 4-3 程序 no\_pingpang 运行结果 time\_A\_LongSon 值

次数	1	2	3	4	5	平均时间 (us)
时间(us)	4013718	4017413	4013953	4020627	4014070	4016091.3
次数	6	7	8	9	10	
时间(us)	4013494	4012064	4016619	4022932	4016023	

表 4-4 程序 has\_pingpang 运行结果 time\_B\_LongSon 值。

次数	1	2	3	4	5	平均时间 (us)
时间(us)	5321858	5351319	5367344	5346380	5306600	5325054.1
次数	6	7	8	9	10	
时间(us)	5306719	5306931	5308861	5322720	5311809	

比较 program\_A 和 program\_B 在龙芯 3 下的平均时间可知, time\_B\_LongSon 的平均值仅略高于 time\_A\_LongSon 平均值, 与 X86 平台下的实验结果比较可知龙芯 3 计算机不存在明显的 cache 乒乓效应。

通过上述严格设计的实验可知, 龙芯 3 计算机 cache 乒乓效应没有 x86 严重。

## 4 实验分析

对比你实验中所用 X86 平台的 cache 结构以及龙芯平台的 cache 结构, 分析为什么龙芯处理器上的 cache 乒乓效应没有这么明显。请考虑各级 cache 在处理器核之间的共享特性。

数据间隔超过 cache 行的间隔就一定不会发生乒乓效应吗？

## 实验 5 访存行为对 cache 性能的影响

### 1 实验目的

- (1) 了解程序访存模式对 cache 利用率产生的差异。
- (2) 掌握有效利用 cache 的编程方法。

### 2 实验原理

由于计算机中数据是按一维线性地址存储数据的，因此连续的两次或多次访问内存能否得到 cache 的帮助，取决于这些访问的物理地址是否都在 cache 中。如果用不同的顺序对数组进行访问，有可能充分利用 cache 的作用，或者无法有效利用 cache 功能。下面是一个简单的程序，用于表明当二维数组分别按行或者列的方式访问，其执行时间的差异。

### 3 实验内容

样例代码非常简单，通过内存分配来为数组获得存储空间，然后分别以逐行或逐列的方式遍历整个数组，并记录各自所需的时间。

执行样例代码，记录程序输出的执行时间结果数据。

#### 样例源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char* argv[])
{
    char **p;
    int i, j, size;
    clock_t start, finish;
    while(~scanf("%d", &size))
    {
        p = (char**)malloc(size * sizeof(char*));
        for(i = 0; i < size; i++)
            p[i] = (char*)malloc(size * sizeof(char));
        /* ....*/
        start = clock();
        for(i = 0; i < size; i++)
            for(j = 0; j < size; j++)
```

```

        p[i][j] = 'b';
    finish = clock();
    printf("%d:", finish - start);
    /* ....*/
    start = clock();
    for(i = 0; i < size; i++)
        for(j = 0; j < size; j++)
            p[j][i] = 'a';
    finish = clock();
    printf("%d\n", finish - start);
    for(i = 0; i < size; i++)
        free(p[i]);
    free(p);
}
return 0;
}

```

## 编译和运行

```

sdbox@loongsonbox-n1:~$ ./cache
1024
40000:40000
2048
180000:180000
3096
400000:420000
^C
sdbox@loongsonbox-n1:~$

```

## 运行结果

表 2 按行和按列访问时间

二维数组大小	128	256	512	1024	2048	4096	8192
按行访问用时(ms)	0	0	20	40	200	820	3320
按列访问用时(ms)	0	0	20	60	260	1440	8240

## 4 实验分析

随着数组越大，相应的执行时间越长，行访问和列访问的时间差距也会越大。

一个主存访问会读取附近的内存进 **cache**，当数组较小时，便可以一次放进 **cache**，此时行列访问的时间差距很小，随着数组的增大，行列访问的时间差距增大，因为行访问对于一次读取的 **cache** 的利用率极高，存储器是按行存储，**cache** 按行读取，程序按行访问，对于列访问来说，随着数组的增大，在未访问完一列的情况下，之前的 **cache** 行被替换的概率越大，导致利用率低，即违背了 **cache** 局部性的原理，故数组小的时候两者差别小而数组大的时候执行时间差别大。

上述程序有没有什么不足？如果分配好空间先遍历一遍会不会更精确地反映

cache 所引起的性能差异——避免 TLB 等其他因素的影响？



## 实验 6 cache 层次及容量评估

### 1 实验目的

- (1) 加深 cache 的层次结构认识。
- (2) 掌握通过控制程序的活动数据集的方法来充分利用 cache 性能。

### 2 实验原理

设计程序不断调整所访问的数据集的大小，逐渐突破 1 级 2 级甚至 3 级 cache 的容量，从而时的各级 cache 出现不同称此的失效，并以此作为各级 cache 容量的估计值。

### 3、 实验内容

阅读和运行程序 `cache.c` 来评估 cache 大小对程序性能的影响。该程序第一个部分是利用系统调用，得到用户精确时间。第二部分是一个嵌套的循环，以不同的步幅和不同活动数据集大小来访问存储器（此段代码运行多次以获得精确的时间间隔）。在样例源代码中设置运行时间为 0.1s，同学们可以设置更长时间以获得更高的精确度。第三部分测量了嵌套循环的时间开销（不带数据访问），以便在全局测量时间中将其减去得到访问时间。在单用户模式下，或者至少在关闭其他程序的情况下运行此程序，能得到更可靠地结果。

#### 样例源代码:

```
#include <stdio.h>
#include <sys/time.h>

#define ARRAY_MIN (1024)    //min cache 4K
#define ARRAY_MAX (1024*1024*4) //max cache 16M
#define SECONDS 0.1

int x[ARRAY_MAX]; //Array going to stride through

double get_seconds() //Routine to read time in seconds
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + (double)tv.tv_usec/1e6;
```

```
}

int label(long i) //For print
{
    if (i < 1e3)
        printf("%ldB\t", i);
    else if (i < 1e6)
        printf("%ldK\t", i/1024);
    else if (i < 1e9)
        printf("%ldM\t", i/1024/1024);
    else
        printf("%ldG\t", i/1024/1024);
    return 0;
}

int main(int argc, char *argv[])
{
    register int nextstep, i, index, stride;
    int csize;
    double steps, tsteps;
    double loadtime, lastsec, sec0, sec1, sec; //time

    printf("\t");
    for (stride = 1; stride <= ARRAY_MAX/2; stride *= 2)
        label(stride*sizeof(int));
    printf("\n");

    for (csize = ARRAY_MIN; csize <= ARRAY_MAX; csize *= 2) //Main loop
    {
        label(csize*sizeof(int)); //Print Cache size this loop
        for (stride = 1; stride <= csize/2; stride *= 2)
        {
            //Lay out path of memory references in array
            for (index = 0; index < csize; index += stride)
                x[index] = index + stride; //Point to next
            x[index-stride] = 0;

            steps = 0.0;
            nextstep = 0;
            sec0 = get_seconds();

            //Walk through path in array for SECONDS seconds
            do
            {
```

```

        for (i = stride; i != 0; i--)
        {
            nextstep = 0;
            do {
                nextstep = x[nextstep];
            } while (nextstep!=0);
        }
        steps += 1.0;
        sec1 = get_seconds();
    } while ((sec1-sec0)<SECONDS);    //set time is SECONDS, the bigger,
the more precise
    sec = sec1-sec0;

    tsteps = 0.0;
    sec0 = get_seconds();

    //Repeat empty loop to loop subtract overhead
    do
    {
        for (i = stride; i != 0; i--)
        {
            index = 0;
            do {
                index += stride;
            } while (index<csiz);
        }
        tsteps += 1.0;
        sec1 = get_seconds();
    } while (tsteps<steps);

    sec -= (sec1-sec0);
    loadtime = (sec*1e9) / (steps*csiz);

    printf("%4.1f\t", (loadtime<0.1) ? 0.1 : loadtime);
}
printf("\n");
}
return 0;
}

```

**编译和运行的方式:**

```

sdbox@loongsonbox-n1:~$ gcc cache_size.c -o cache_size
sdbox@loongsonbox-n1:~$ ./cache_size

```

	4B	8B	16B	32B	64B	128B	256B	512B	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M
4K	9.4	9.4	9.4	9.4	9.4	9.3	9.3	9.1	8.5	0.1												
8K	9.4	9.4	9.4	9.4	9.4	9.4	9.3	9.3	9.1	8.5	1.9											
16K	9.4	9.4	9.4	9.4	9.4	9.4	9.4	9.4	9.3	9.1	8.5	0.1										
32K	9.4	9.4	9.4	9.4	9.4	9.4	9.4	9.4	9.3	9.1	8.5	0.1										
64K	9.6	9.6	9.6	9.6	9.6	9.5	9.4	9.4	9.4	9.3	9.1	8.5	0.8									
128K	10.9	12.4	15.4	21.3	33.2	33.2	33.2	33.2	33.2	33.2	33.2	33.1	8.5	0.1								
256K	10.9	12.4	15.4	21.3	33.2	33.3	33.2	33.2	33.2	33.2	33.2	33.2	33.1	8.5	0.1							
512K	10.4	11.3	13.1	16.8	24.0	73.6	73.4	73.3	73.3	73.3	73.3	73.3	73.3	33.2	33.0	8.5	0.1					
1M	10.4	11.3	13.2	17.0	24.5	73.6	73.4	73.4	73.3	73.4	73.3	73.3	73.3	33.2	33.1	8.5	0.1					
2M	10.4	11.3	13.1	16.8	24.1	73.5	73.5	73.4	73.5	73.7	74.1	74.8	76.9	79.3	73.3	33.2	33.1	8.4	0.1			
4M	10.3	11.2	13.1	16.7	24.1	132.4	132.5	132.9	133.3	134.8	137.9	146.0	149.1	100.5	79.3	33.2	33.1	8.5	0.1			
8M	10.3	11.2	13.0	16.6	23.8	199.8	200.3	201.0	202.5	205.5	211.2	224.7	228.2	223.1	214.7	204.6	181.9	33.2	33.1	8.5	1.9	
16M	10.3	11.2	13.0	16.5	23.6	199.7	200.2	200.9	202.5	205.4	211.1	224.6	228.0	227.7	223.7	219.0	209.2	200.5	33.2	33.1	8.5	0.1

```

sdbox@loongsonbox-n1:~$

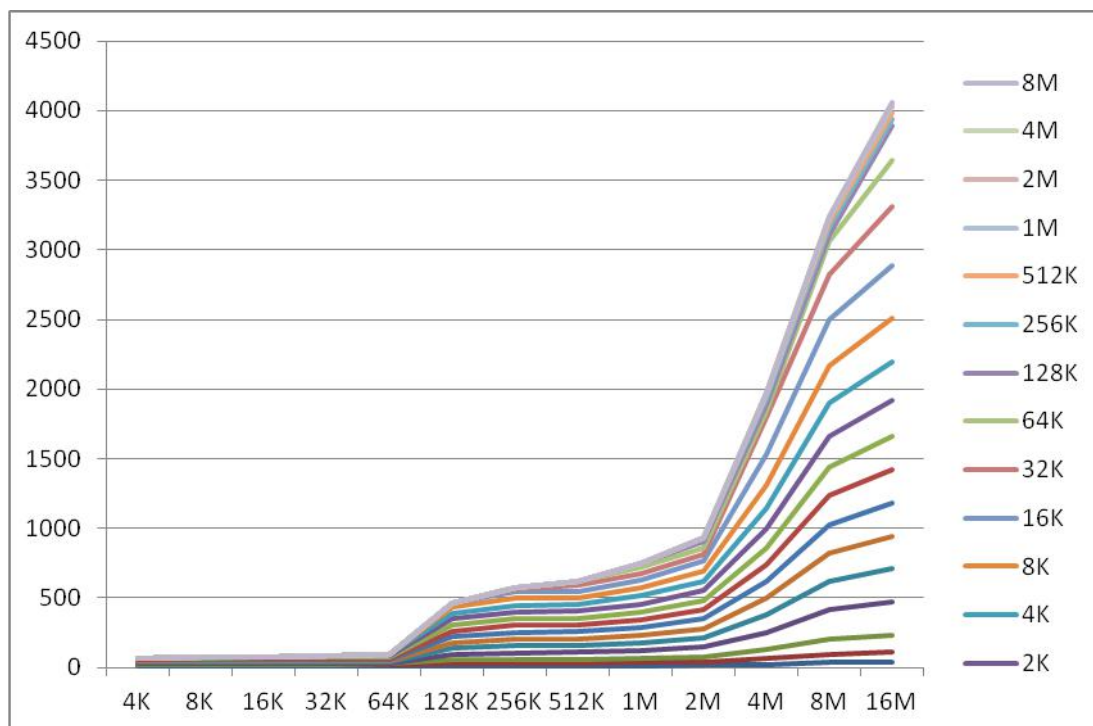
```

图 6-1 运行结果输出

### 实验结果:

龙芯处理器的 cache 配置数据如下: cache: L1: 64KB 数据; 64KB 代码; L2: 4MB。

将图 6-1 的数据绘制曲线如图所示, 横坐标为活动数据集, 纵坐标为平均访问延迟, 不同曲线对应不同的数据访问步幅。



## 4 实验分析

在图中, 水平轴表示的是工作集的大小 (Working set size), 纵轴是时间, 单位是 s, 从折线中可以看出在工作集大小为 64KB 和 2M~8M 之间有较大的波动。故可以推断分别为 L1, L2Cache 的大小, 思考为什么可以这样推断, 并且为什么所测到的是在 2M~8M 之间程序性能有较大的影响, 而不是在 L2Cache 实际大小 4M 的地方。

该程序在当前主流 x86 平台上运行所得的结果曲线可以看出三个台阶, 对应于几级 cache 的层次?

## 实验 7 TLB 命中与缺失对访存性能的影响

### 1 实验目的

测试龙芯处理器TLB命中与失效情况下的访存时间比。

### 2 实验原理

TLB: Translation lookaside buffer, 即旁路转换缓冲, 又称为快表技术。快表是一块小容量的相联存储器 (Associative Memory), 由高速缓存器组成, 速度快, 并且可以从硬件上保证按内容并行查找, 一般用来存放当前访问最频繁的少数活动页面的页号。

快表的用途是加快线性地址的转换。当一个线性地址第一次使用时, 通过慢速访问 RAM 中的页表计算出相应的物理地址。同时, 物理地址被存放在一个 TLB 表项中, 以便以后对同一个线性地址的引用可以快速得到转换。

如果 TLB 中正好存放着所需的页表, 则称为 TLB 命中 (TLB Hit); 如果 TLB 中没有所需的页表, 则称为 TLB 失败 (TLB Miss)。

### 3 实验内容

阅读样例代码, 编译执行并记录输出结果数据。

#### 程序框架

向系统申请一块足够大的内存, 然后依次访问内存, 以便建立起虚实影射防止缺页的情况。然后再按跨 cache 行 (排除 cache 影响) 和跨页两种方式访问申请到的内存, 得到访问的耗时  $T1$  和  $T2$ 。最后再比较  $T1$  和  $T2$  的大小, 如果  $T2$  明显大于  $T1$ , 就说明 TLB 缺失已经对程序产生的较大的影响。

#### 样例代码

tlb\_miss.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
```

```

#define LOOPS 256*1024
#define CACHE_LINE_SIZE 32
#define PAGE_SIZE 4096
#define MEM_SIZE (PAGE_SIZE*LOOPS)

int main(int argc, char *argv[])
{
    struct timeval tpStart, tpEnd;
    float t0, timeUse = 0;

    int i;
    char ch;
    char *arr = malloc(MEM_SIZE);
    for (i = 0; i < LOOPS; ++i)
        arr[i*PAGE_SIZE]=i;

    gettimeofday(&tpStart, NULL);
    for (i = 0; i < LOOPS; ++i)
        ch = 0;
    gettimeofday(&tpEnd, NULL);
    t0 =
1000000.*(tpEnd.tv_sec-tpStart.tv_sec)+tpEnd.tv_usec-tpStart.tv_usec;

    gettimeofday(&tpStart, NULL);
    for (i = 0; i < LOOPS; ++i)
        ch = arr[i*CACHE_LINE_SIZE];
    gettimeofday(&tpEnd, NULL);
    timeUse =
1000000.*(tpEnd.tv_sec-tpStart.tv_sec)+tpEnd.tv_usec-tpStart.tv_usec-t0;
    printf("the cost of total time is %f us.\n", timeUse);

    gettimeofday(&tpStart, NULL);
    for (i = 0; i < LOOPS; ++i)
        ch = arr[i*PAGE_SIZE];
    gettimeofday(&tpEnd, NULL);
    timeUse =
1000000.*(tpEnd.tv_sec-tpStart.tv_sec)+tpEnd.tv_usec-tpStart.tv_usec-t0;
    printf("the cost of total time is %f us (TLB missing).\n", timeUse);

    free(arr);
    return 0;
}

```

**编译和运行的方式：**

```

sdbox@loongsonbox-n1:~$ gcc tlb_miss.c
sdbox@loongsonbox-n1:~$ ./a.out
the cost of total time is 4591.000000 us.
the cost of total time is 61231.000000 us (TLB missing).
sdbox@loongsonbox-n1:~$

```

### 实验结果:

#### X86 平台的运行结果

时间(us)	1	2	3	4	5	6	平均值
TLB命中	862	806	803	826	829	656	797
TLB失败	4509	4499	4475	4506	4518	4346	4476

#### Loongson 3 平台的运行结果

时间(us)	1	2	3	4	5	6	平均值
TLB命中	72625	74802	73827	74941	74155	74987	74223
TLB失败	79008	79179	78923	78913	79099	78925	79008

注：龙芯 TLB Entries为64

相对比与 x86 的结果，我们可以发现龙芯平台下 TLB 对访存性能的影响并没有 x86 平台那么明显。

## 4 实验分析

为何需要用跨 cache 行的访问方式，如果第一个程序是按照逐字方式访问会有什么不妥？如果在开始两种访问之前不遍历访问一次全部空间，结果将有什么变化？



## 实验 8 内存控制器带宽与竞争

### 1 实验目的.

- (1) 体验主存带宽竞争的存在。
- (2) 加深主存带宽竞争程序访存的影响的认识。

### 2 实验原理

在四核的龙芯 3A 处理器上使用不同的线程数目（1~4）访问内存数据，各个线程分别绑定到不同的处理器核上（同一 NUMA 节点中，即同一 CPU 封装内）。访问数据时为了避免 cache 映像、使得每次访问都能到达内存控制器，访问间隔要大于 cache 行。

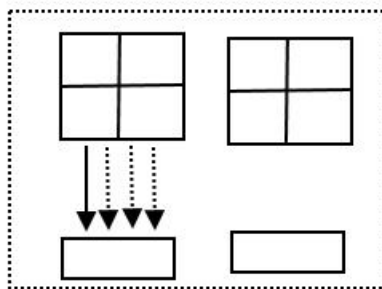


图 8-1 多线程竞争内存控制器示意图

### 3 实验内容

样例程序说明：

样例程序中，首先从系统中获取系统的 cache 行大小赋值个变量 `cache_line_size`，如果获取失败，默认大小为 64B，然后构建一个 1GB 大小的数组，并进行初始化，统计 openmp 多线程跳过 cache 行跳读访问整个数组时间，打印输出。

样例代码：imc\_competition.c

```
#include<stdlib.h>
#include<stdio.h>
#include<sys/time.h>
#define MEM_SIZE (1024*1024*1024)    //1GB
```

```

int main(int argc, char *argv[])
{
    /*获取cache行大小*/
    int cache_line_size=64;
    char ch;
    int i;
    char temp[3];
    struct timeval start,end;
    FILE
*cachefile=fopen("/sys/devices/system/cpu/cpu0/cache/index3/coherency_line_s
ize","r");
    if(cachefile)
    {
        fgets(temp,3,cachefile);
        cache_line_size=atoi(temp);
    }
    else
        printf("cant not get get cach line size,default
is %dB\n",cache_line_size);
    // fclose(cachefile);
    char *mem=(char *)malloc(MEM_SIZE);
    for(i=0;i<MEM_SIZE;i++)
    {
        mem[i]='a';
    }
    //读内存数据，跳过cache
    gettimeofday(&start,NULL);
    #pragma omp parallel for
    for(i=0;i<MEM_SIZE;i=i+cache_line_size)
    {
        ch= mem[i];
    }
    gettimeofday(&end,NULL);
    int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec
-start.tv_usec;
    printf("read %d MB data from mem, time used:%d
USEC\n", (MEM_SIZE/cache_line_size)/(1024*1024),timeuse);
    free(mem);
    return 0;
}

```

## 编译与运行

编译命令如下：

```

sdbox@loongsonbox-n1:~$ gcc -fopenmp imc_competition.c -o imc_competition

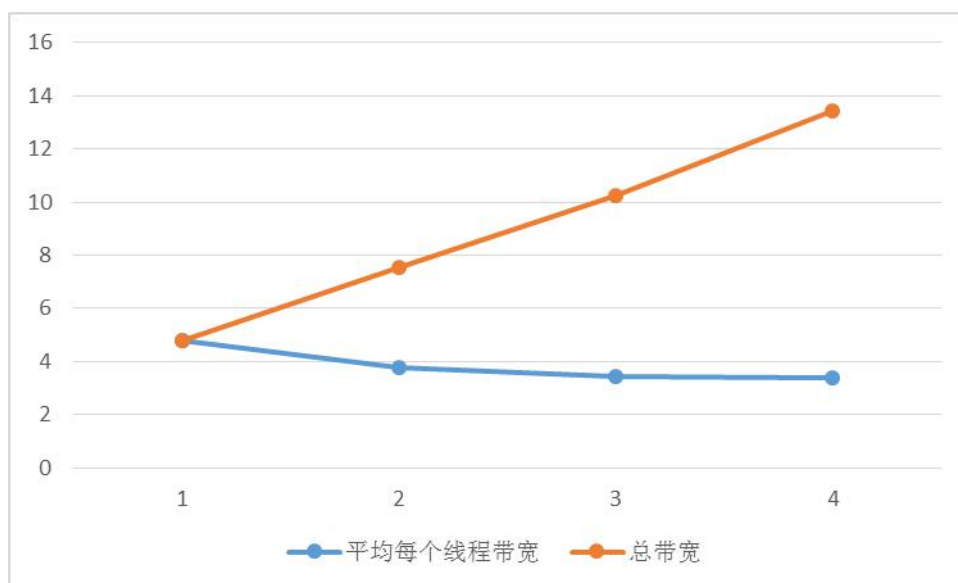
```

运行时，我们通过 NUMACTL 工具将程序的内存数据绑定在本地的 0 号上，然后每次运行增加线程数目，使产生多个线程对 0 号主存访问带宽的竞争情况。

```
sdbox@loongsonbox-n1:~$ numactl --physcpubind=0 --membind=0 ./imc_competition
cant not get get cach line size,default is 64B
read 16 MB data from mem, time used:805655 USEC
sdbox@loongsonbox-n1:~$ numactl --physcpubind=0,1 --membind=0 ./imc_competition
cant not get get cach line size,default is 64B
read 16 MB data from mem, time used:762720 USEC
sdbox@loongsonbox-n1:~$ numactl --physcpubind=0,1,2 --membind=0 ./imc_competition
cant not get get cach line size,default is 64B
read 16 MB data from mem, time used:623335 USEC
sdbox@loongsonbox-n1:~$ numactl --physcpubind=0,1,2,3 --membind=0 ./imc_competition
cant not get get cach line size,default is 64B
read 16 MB data from mem, time used:575261 USEC
sdbox@loongsonbox-n1:~$
```

### 实验结果

分别统计一个、两个、三个和四个线程对 0 号主存同时访问时的总带宽和平均每个线程的带宽趋势如下图所示。



从上图中我们可以看出随着线程数组的增加，程序访问主存的总带宽成线性趋势增长，但是平均每个线程的带宽，由于受到其他线程对内存控制器竞争的影响，成降低的趋势。

## 4 实验分析

根据龙芯 3A 的内存带宽变化趋势，请分析处理器访存能力与内存控制器的服务能力之间的匹配情况。

样例程序中各个线程访问的是同一内存区间，这样的共享数据访问会不会对内存带宽测试有影响？有无必要访问各自不相交内存空间？

考虑到各个线程执行时间会受到调度的影响，也就是说并不是所有线程在严格同步的时间点上开始执行，这对测试结果有什么影响

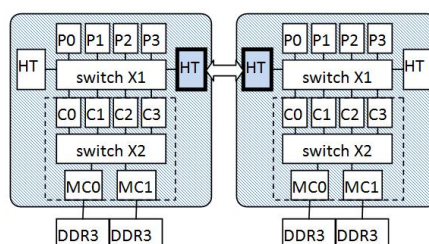
## 实验 9 访存中的 NUMA 因素

### 1 实验目的

- (1) 形成 NUMA 架构的直观认识。
- (2) 体验访存过程中 NUMA 因素的存在。
- (3) 掌握通过系统工具控制 NUMA 因素的方法。

### 2 实验背景与原理

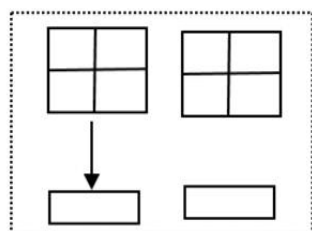
PC 服务器处理器/核数随着需求的提高而不断的增长,使得 SMP 结构的内存冲突显得更严重,于是采用了 NUMA 内存的结构。两颗龙芯 3A 处理器通过 HT 互联构成 8 核 NUMA 系统如下图所示;



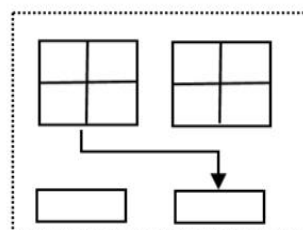
如上图所示 NUMA 系统上,处理器访问非本地内存(附加到另一个处理器)的内存需要经过 HT 总线,因此处理器能够访问非本地内存要比访问本地内存(直接附加到该处理器的内存)慢,内存相对于处理器的位置对处理器访问内存所需的时间的影响,称为 NUMA 因素。

### 3 实验内容

编写一个访存的程序,使用 NUMACTL 工具,将程序的数据分别绑定在本地内存和远地内存上,比较程序在这两种情况下的访存性能差异。访存模型如下图所示。



访问本地内存



访问远地内存

图 9-1 本地访问与远程访问的示意图

样例程序中，首先从系统中获取系统的 cache 行大小赋值个变量 `cache_line_size`，如果获取失败，默认大小为 64B，然后构建一个 1GB 大小的数组，并进行初始化，然后以 cache 行大小的幅度跳读整个数组元素（确保每次访问都是从主存中获取数据），最后统计程序访存时间，打印输出。

#### 样例程序源码：numa\_factor.c

```
#include<stdlib.h>
#include<stdio.h>
#include<sys/time.h>
#define MEM_SIZE (1024*1024*1024)    //1GB
int main(int argc, char *argv[])
{
    /*获取cache行大小*/
    int cache_line_size=64;
    char ch;
    int i;
    char temp[3];
    struct timeval start,end;
    FILE
*cachefile=fopen("/sys/devices/system/cpu/cpu0/cache/index3/coherency_line_s
ize","r");
    if(cachefile)
    {
        fgets(temp,3,cachefile);
        cache_line_size=atoi(temp);
    }
    else
        printf("cant not get get cach line size,default
is %dB\n",cache_line_size);
    // fclose(cachefile);
    char *mem=(char *)malloc(MEM_SIZE);
    for(i=0;i<MEM_SIZE;i++)
    {
        mem[i]='a';
    }
    //读内存数据，跳过cache
    gettimeofday(&start,NULL);
    for(i=0;i<MEM_SIZE;i=i+cache_line_size)
    {
        ch= mem[i];
```

```

    }
    gettimeofday(&end, NULL);
    int timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec
- start.tv_usec;
    printf("read %d MB data from mem, time used:%d
USEC\n", (MEM_SIZE/cache_line_size)/(1024*1024), timeuse);
    free(mem);
    return 0;
}

```

### 编译与运行

我们通过 NUMACTL 工具将程序运行的处理器核绑定在 0 号处理器核上，程序的内存数据分别绑定在本地的 0 号内存上（本地内存）和 1 号内存上（非本地内存），来运行程序。

```

sdbox@loongsonbox-n1:~$ gcc -Wall -g numa_factor.c -o numa_factor
sdbox@loongsonbox-n1:~$ numactl --physcpubind=0 --membind=0 ./numa_factor
cant not get get cach line size,default is 64B
read 16 MB data from mem, time used:798671 USEC
sdbox@loongsonbox-n1:~$ numactl --physcpubind=0 --membind=1 ./numa_factor
cant not get get cach line size,default is 64B
read 16 MB data from mem, time used:1159544 USEC
sdbox@loongsonbox-n1:~$

```

### 实验结果

从程序的打印结果可以看出，访问本地内存时间大概 0.79 秒，访问远地内存时间大概 1.1 秒，访问远地内存的速度要比本地内存差不多慢一倍。

## 4 实验分析

考虑 4 个节点共 16 核的龙芯 NUMA 平台，将计算线程与数据绑定到不同的节点上，将会产生多少种不同的访问延迟？结合龙芯 16 核平台的互连结构，测试具体的访问延迟数据并作分析。



## 实验 10 集群结构体验——MPI 编程

### 1 实验目的

- (1) 形成集群架构的直观认识。
- (2) 了解分布式内存结构上的消息传递内存编程模式。
- (3) 初步了解 MPI 编程。

### 2 实验背景

多线程是一种便捷的模型，其中每个线程都可以访问其它线程的存储空间。因此，这种模型只能在共享存储系统之间移植。集群计算机没有共享存储，当面向非共享存储系统开发并行程序时，程序的各部分之间通过来回传递消息的方式通信。要使得消息传递方式可移植，就需要采用标准的消息传递库。这就促成的消息传递接口(Message Passing Interface, MPI)的面世，MPI 是一种被广泛采用的消息传递标准。

与 OpenMP 并行程序不同，MPI 是一种基于消息传递的并行编程技术。消息传递接口是一种编程接口标准，而不是一种具体的编程语言。简而言之，MPI 标准定义了一组具有可移植性的编程接口。各个厂商或组织遵循这些标准实现自己的 MPI 软件包，典型的实现包括开放源代码的 MPICH、LAM MPI 以及不开放源代码的 Intel MPI。由于 MPI 提供了统一的编程接口，程序员只需要设计好并行算法，使用相应的 MPI 库就可以实现基于消息传递的并行计算。MPI 支持多种操作系统，包括大多数的类 UNIX 和 Windows 系统。

MPI 程序是基于消息传递的并行程序。消息传递指的是并行执行的各个进程具有自己独立的堆栈和代码段，作为互不相关的多个程序独立执行，进程之间的信息交互完全通过显式地调用通信函数来完成。

### 3 实验内容

#### Cannon 算法

本实验用 MPI 完成 Cannon 算法的并行计算。Cannon 算法的基本思想可以如

下表示：假设两个矩阵 A 和 B 相乘，把 A 和 B 矩阵划分成  $p$  个方块，进程的编号从  $p_0$  到  $p_{\sqrt{p}-1}$ ，并在最初把子矩阵  $A_{i,j}$  和  $B_{i,j}$  分配给  $P_{i,j}$ 。虽然第  $i$  行的每个进程需要全部  $\sqrt{p}$  的个子矩阵  $A_{i,k}$ ，但我们还是能调度第  $i$  行  $\sqrt{p}$  个进程的计算，使得每个进程在任何时刻都是用不同的  $A_{i,k}$ 。每完成一次矩阵乘法，这些块在各进程之间被轮流使用，似的每次轮流之后每个进程都可以得到新的  $A_{i,k}$ 。对列使用同样的调度，则在任何时刻，任何进程至多拥有每个矩阵的一个块，在所有进程中，改算法需要的总内存量为  $n^2$ 。下图为此算法中不同进程上子矩阵乘法的调度过程。

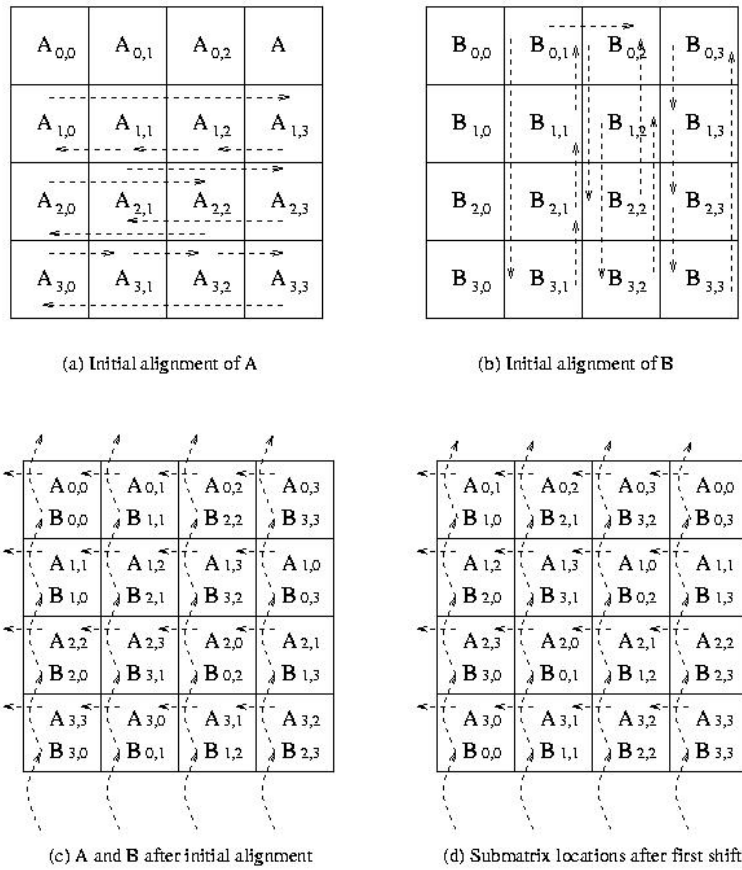


图 11-1 cannon 算法子矩阵乘法的调度过程

假如矩阵  $C=A*B$ , 则 C 的 的计算公式如下:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{ik} B_{kj} \quad (0 \leq i \leq \sqrt{p}-1, 0 \leq j \leq \sqrt{p}-1)$$

进程 P 存储分块矩阵这一部分。块矩阵乘法要计算所有匹配的  $A_{ik}$  和  $B_{kj}$ ，然

而只有在主对角线的才是匹配的。因此需要采用循环移动分块矩阵的方法来使每个进程 $P_{ij}$ 都有一对可以直接相乘的匹配的块，具体方法如下：

(1)将排第  $i$  行的 $A_{ik}$ 块循环左移  $i$  个位置，将第 $B_{kj}$ 列 块循环上移  $j$  个位置；

(2) 进程 $P_{ij}$ 执行乘一加运算，然后将移动得到的 $A_{ik}$  块循环左移 1 个位置，将移动得到的  $B_{kj}$ 块循环上移 1 个位置；

(3)重复第 2 步( $\sqrt{p}-1$ )次，每次移动后 $P_{ij}$ 进程执行乘一加运算。

经过以上操作后就可以得到矩阵  $C$  的解。

### 实验代码

```
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <time.h>
#include <stdio.h>
#include <math.h>

#define MAX_NUMBER 10

/* 全局变量声明 */
float **A, **B, **C;          /* 总矩阵, C = A * B */
float *a, *b, *c, *tmp_a, *tmp_b; /* a、b、c表分块, tmp_a、tmp_b表缓冲区 */
int dg, dl, dl2, p, sp;       /* dg:总矩阵维数;dl:矩阵块维数;dl2=dl*dl;p:处理器个数;sp=sqrt(p) */
int my_rank, my_row, my_col;  /* my_rank:处理器ID;(my_row,my_col):处理器逻辑阵列坐标 */
MPI_Status status;

/*
 * 函数名: get_index
 * 功能: 处理器逻辑阵列坐标至rank号的转换
 * 输入: 坐标、逻辑阵列维数
 * 输出: rank号
 */
int get_index(int row, int col, int sp)
{
    return ((row + sp) % sp) * sp + (col + sp) % sp;
}
```

```

/*
 * 函数名: random_A_B
 * 功能: 随机生成矩阵A和B
 */
void random_A_B()
{
    int i, j;

    srand((unsigned int)time(NULL));    /* 设随机数种子 */

    /* 随机生成A、B,并初始化C */
    for (i = 0; i < dg; i++)
        for (j = 0; j < dg; j++)
        {
            A[i][j] = (float)rand()/rand();
            B[i][j] = (float)rand()/rand();
            C[i][j] = 0.0;
        }
}

/* 函数名: scatter_A_B
   功能: rank为0的处理器向其他处理器发送A、B矩阵的相关块 */
void scatter_A_B()
{
    int i, j, k, l;
    int p_imin, p_imax, p_jmin, p_jmax;

    for (k = 0; k < p; k++)
    {
        /* 计算相应处理器所分得的矩阵块在总矩阵中的坐标范围 */
        p_jmin = (k % sp) * dl;
        p_jmax = (k % sp + 1) * dl - 1;
        p_imin = (k - (k % sp)) / sp * dl;
        p_imax = ((k - (k % sp)) / sp + 1) * dl - 1;
        l = 0;

        /* rank=0的处理器将A、B中的相应块拷至tmp_a, tmp_b, 准备向其他处理器发送
        */
        for (i = p_imin; i <= p_imax; i++)
        {
            for (j = p_jmin; j <= p_jmax; j++)
            {

```

```

        tmp_a[l] = A[i][j];
        tmp_b[l] = B[i][j];
        l++;
    }
}

/* rank=0的处理器直接将自己对应的矩阵块从tmp_a, tmp_b拷至a, b */
if (k == 0)
{
    memcpy(a, tmp_a, dl2 * sizeof(float));
    memcpy(b, tmp_b, dl2 * sizeof(float));
}
else /* rank=0的处理器向其他处理器发送tmp_a, tmp_b
中相关的矩阵块 */
{
    MPI_Send(tmp_a, dl2, MPI_FLOAT, k, 1, MPI_COMM_WORLD);
    MPI_Send(tmp_b, dl2, MPI_FLOAT, k, 2, MPI_COMM_WORLD);
}
}

/*
*函数名:init_alignment
*功能:矩阵A和B初始对准
*/
void init_alignment()
{
    /* 将A中坐标为(i, j)的分块A(i, j)向左循环移动i步 */
    MPI_Sendrecv(a, dl2, MPI_FLOAT, get_index(my_row, my_col - my_row, sp), 1,
        tmp_a, dl2, MPI_FLOAT, get_index(my_row, my_col + my_row, sp), 1,
        MPI_COMM_WORLD, &status);
    memcpy(a, tmp_a, dl2 * sizeof(float));

    /* 将B中坐标为(i, j)的分块B(i, j)向上循环移动j步 */
    MPI_Sendrecv(b, dl2, MPI_FLOAT, get_index(my_row - my_col, my_col, sp), 1,
        tmp_b, dl2, MPI_FLOAT, get_index(my_row + my_col, my_col, sp), 1,
        MPI_COMM_WORLD, &status);
    memcpy(b, tmp_b, dl2 * sizeof(float));
}

/*
*函数名: main_shift
*功能: 分块矩阵左移和上移, 并计算分块c
*/

```

```

void main_shift()
{
    int i, j, k, l;

    for (l = 0; l < sp; l++)
    {
        /* 矩阵块相乘, c+=a*b */
        for (i = 0; i < dl; i++)
            for (j = 0; j < dl; j++)
                for (k = 0; k < dl; k++)
                    c[i * dl + j] += a[i * dl + k] * b[k * dl + j];

        /* 将分块a左移1位 */
        MPI_Send(a, dl2, MPI_FLOAT, get_index(my_row, my_col - 1, sp), 1,
                 MPI_COMM_WORLD);
        MPI_Recv(a, dl2, MPI_FLOAT, get_index(my_row, my_col + 1, sp), 1,
                 MPI_COMM_WORLD, &status);

        /* 将分块b上移1位 */
        MPI_Send(b, dl2, MPI_FLOAT, get_index(my_row - 1, my_col, sp), 1,
                 MPI_COMM_WORLD);
        MPI_Recv(b, dl2, MPI_FLOAT, get_index(my_row + 1, my_col, sp), 1,
                 MPI_COMM_WORLD, &status);
    }
}

/*
 * 函数名: collect_c
 * 功能: rank为0的处理器从其余处理器收集分块矩阵c
 */
void collect_C()
{
    int i, j, i2, j2, k;
    int p_imin, p_imax, p_jmin, p_jmax; /* 分块矩阵在总矩阵中顶点边界值 */

    /* 将rank为0的处理器中分块矩阵c结果赋给总矩阵C对应位置 */
    for (i = 0; i < dl; i++)
        for (j = 0; j < dl; j++)
            C[i][j] = c[i * dl + j];

    for (k = 1; k < p; k++)
    {
        /* 将rank为0的处理器从其他处理器接收相应的分块c */
        MPI_Recv(c, dl2, MPI_FLOAT, k, 1, MPI_COMM_WORLD, &status);
    }
}

```

```

    p_jmin = (k % sp) * dl;
    p_jmax = (k % sp + 1) * dl - 1;
    p_imin = (k - (k % sp)) / sp * dl;
    p_imax = ((k - (k % sp)) / sp + 1) * dl - 1;

    i2 = 0;
    /* 将接收到的c拷至C中的相应位置,从而构造出C */
    for (i = p_imin; i <= p_imax; i++)
    {
        j2 = 0;
        for (j = p_jmin; j <= p_jmax; j++)
        {
            C[i][j] = c[i2 * dl + j2];
            j2++;
        }
        i2++;
    }
}

/*
 * 函数名: print
 * 功能: 打印矩阵
 * 输入: 指向矩阵指针的指针, 字符串
 */
void print(float **m, char *str)
{
    int i, j;

    printf("%s", str);
    /* 打印矩阵m */
    for (i = 0; i < dg; i++)
    {
        for (j = 0; j < dg; j++)
            printf("%.4f\t", m[i][j]);
        printf("\n");
    }
    printf("\n");
}

/*
 * 函数名: main
 * 功能: 主过程, Cannon算法, 矩阵相乘

```



```

* 输入: argc为命令行参数个数, argv为每个命令行参数组成的字符串数组
*/
int main(int argc, char *argv[])
{
    int i;

    MPI_Init(&argc, &argv);    /* 启动MPI计算 */
    MPI_Comm_size(MPI_COMM_WORLD, &p); /* 确定处理器个数 */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /* 确定各自的处理器标识符 */

    sp = sqrt(p);

    /* 确保处理器个数是完全平方数, 否则打印错误信息, 程序退出
    */
    if (sp * sp != p)
    {
        if (my_rank == 0)
            printf("Number of processors is not a quadratic number!\n");
        MPI_Finalize();
        exit(1);
    }

    if (argc != 2)
    {
        if (my_rank == 0)
            printf("usage: mpirun -np ProcNum %s MatrixDimension\n", argv[0]);
        MPI_Finalize();
        exit(1);
    }

    dg = atoi(argv[1]);    /* 总矩阵维数 */
    dl = dg / sp;          /* 计算分块矩阵维数 */
    dl2 = dl * dl;

    /* 计算处理器在逻辑阵列中的坐标 */
    my_col = my_rank % sp;
    my_row = (my_rank - my_col) / sp;

    /* 为a、b、c分配空间 */
    a = (float *)malloc(dl2 * sizeof(float));
    b = (float *)malloc(dl2 * sizeof(float));
    c = (float *)malloc(dl2 * sizeof(float));

    /* 初始化c */

```

```

for (i = 0; i < dl2; i++)
    c[i] = 0.0;

/* 为tmp_a、tmp_b分配空间 */
tmp_a = (float *)malloc(dl2 * sizeof(float));
tmp_b = (float *)malloc(dl2 * sizeof(float));

if (my_rank == 0)
{
    /* rank为0的处理器为A、B、C分配空间 */
    A = (float **)malloc(dg * sizeof(float *));
    B = (float **)malloc(dg * sizeof(float *));
    C = (float **)malloc(dg * sizeof(float *));

    for (i = 0; i < dg; i++)
    {
        A[i] = (float *)malloc(dg * sizeof(float));
        B[i] = (float *)malloc(dg * sizeof(float));
        C[i] = (float *)malloc(dg * sizeof(float));
    }
    random_A_B();          /* rank为0的处理器随机化生成A、B矩阵 */
    scatter_A_B();          /* rank为0的处理器向其他处理器发送A、B矩阵的
相关块 */
}
else                      /* rank不为0的处理器接收来自rank为0的处理器
的相应矩阵分块 */
{
    MPI_Recv(a, dl2, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(b, dl2, MPI_FLOAT, 0, 2, MPI_COMM_WORLD, &status);
}

init_alignment();          /* A、B矩阵的初始对准 */

main_shift();              /* 分块矩阵左移、上移，cannon算法的主过程 */

if (my_rank == 0)
{
    collect_C();            /* rank为0的处理器从其余处理器收集分块矩阵c
*/

    print(A, "random matrix A : \n");      /* 打印矩阵A */
    print(B, "random matrix B : \n");      /* 打印矩阵B */
    print(C, "Matrix C = A * B : \n");     /* 打印矩阵C */
}

```

```

else
{
    MPI_Send(c, d12, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);    /* rank不为0
的处理器向rank为0的处理器发送矩阵块c */
}

MPI_Barrier(MPI_COMM_WORLD);    /* 同步所有处理器 */
MPI_Finalize();    /* 结束MPI计算 */

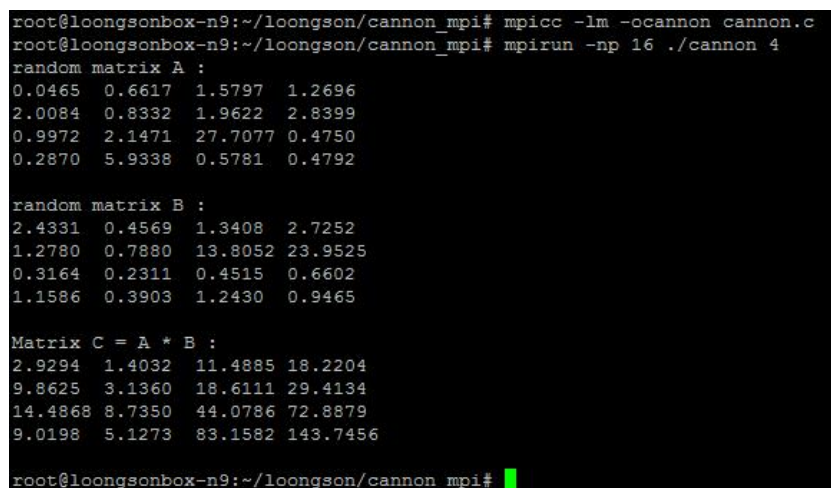
return 0;
}

```

### 编译和运行 MPI 程序

编译 MPI 程序必须用 `mpicc`，编译参数和 `gcc` 一样。

而运行程序的时候使用 `mpirun` 命令，通过 `-np` 参数来控制处理器的数量，其中 4 为要计算的矩阵的维度，设为  $\sqrt{np}$  的整数倍。



```

root@loongsonbox-n9:~/loongson/cannon_mpi# mpicc -lm -ocannon cannon.c
root@loongsonbox-n9:~/loongson/cannon_mpi# mpirun -np 16 ./cannon 4
random matrix A :
0.0465  0.6617  1.5797  1.2696
2.0084  0.8332  1.9622  2.8399
0.9972  2.1471  27.7077  0.4750
0.2870  5.9338  0.5781  0.4792

random matrix B :
2.4331  0.4569  1.3408  2.7252
1.2780  0.7880  13.8052  23.9525
0.3164  0.2311  0.4515  0.6602
1.1586  0.3903  1.2430  0.9465

Matrix C = A * B :
2.9294  1.4032  11.4885  18.2204
9.8625  3.1360  18.6111  29.4134
14.4868  8.7350  44.0786  72.8879
9.0198  5.1273  83.1582  143.7456

root@loongsonbox-n9:~/loongson/cannon_mpi#

```

图 11-2

## 4 实验分析

在 4 路 4 核龙芯的配置环境下，调整 MPI 进程数量（1~16），计算加速比，分析通信在 MPI 程序中对加速比的影响。

注意 MPI 中消息传递的使用，并对比实验 10 中 SMP 结构中使用共享内存的差异。

## 附录 1 实验报告格式

# 实 验 报 告

学生姓名： XXX      学 号： XXX      指导教师： XXX

实验地点： XX 实验室      实验时间： YY/MM/DD

一、 实验室名称：

二、 实验项目名称：

三、 实验目的：

四、 必修或选修：

五、 实验平台：

六、 实验内容及步骤：

根据不同的实验内容，写出具体的实验步骤。

七、 实验数据及结果分析：

根据不同的实验内容，记录具体的实验数据或程序运行结果。实验数据量较大时，最好制成表格形式。程序设计实验，要以电子文档形式附上程序功能、模块说明、完整源代码，源代码中尽量多带注释；阐述对程序性能改进的具体方法及主要调试过程。

八、 实验结论及总结：