University of Science and Technology of China

基于 Inferno 的集群与并行计算框架

阮震元　解宇飞　杨智　刘旭彤*

阮震元：PB13011009, 联系方式 rzyrzyrzy2014@gmail.com, 科技实验楼 1406 系统设计室
解宇飞：PB13011001　杨智：PB13011079　刘旭彤：PB13011072

Contents

1 可行性研究报告

1.1 理论依据与技术依据

1.1.1 Sytx 协议及其具体实现

Inferno 的文件系统协议（或 Sytx）是贝尔实验室九号项目所开发的针对于分布式操作系统的网络协议，作用是链接系统内的组件.Inferno 系统中文件是核心，这些文件代表了视窗、网络连接、进程、以及其他存在于操作系统中的各种元素. 不同于 NFS，Styx 的用途是把数据缓存起来，并提供虚拟文件的机制（例如/proc 用以表示进程）.

在 Inferno 系统中，Inferno 服务器（Inferno server）是被用来被 Inferno 进程访问的并提供分等级文件系统（文件树）的代理. 一个服务器响应客户端的请求并导引文件树中的具体位置，同事可以对文件进行创建、删除、读写等操作.

对于某个服务器来说，连接是通过一个客户端和服务器间的双向的通信通道来实现的. 客户端可能是单一的，也可能有多个客户共享同一连接. 一个服务器文件的文件树通过 bind 和 mount 命令联结到一个进程组的命名空间上. 这样，组内进程就是服务器的客户端：对于文件的系统调用就被翻译为请求和响应，并得到相应的服务.

Inferno 的文件协议，Sytx，就是用于处理客户端和服务器间的消息（message）的. 目前我们组所用的 Sytx 版本是和 9P2000 一致的. 客户端给服务器传递请求（T-messages）. 相应的，服务器返回响应给客户端（R-messages）. 这样就构成了基本的发送和接收机制. 关于 message 的样式，这里就不再赘述. 不过值得一提的是 fid——一个客户端用来辨认服务器上当前文件的 32 位无符号整型.

Styx 在客户端及服务器端提交如下的消息. 这些消息对应到虚拟文件系统层的进入点，所有的服务器都必须实现这些消息：

1. version：交涉协议的版本.
2. error：error.
3. flush：终止消息.
4. auth, attach：打开连接.
5. walk：走访目录层次结构.

6. create, open：准备一个用来写入/读取既有或新增文件的 fid.
7. read, write：发送数据给文件或从文件接收数据.
8. clunk：抛弃 fid.
9. remove：从服务器移除文件.
10. stat, wstat：查询或变更文件属性.

这些具体实现中，有几个是我们工程的技术依据：

1. walk message：walk 消息使得服务器变更当前的与 fid 相联系的文件为一个在目录中旧的“当前文件”或者某一个它的子目录.Walk 返回一个新的 fid，这个 fid 指向于该文件. 通常，客户为根目录保留 fid，然后通过 walk 从根 fid 进行导引.
2. attach，auth message：attach 消息用来从客户端的用户向服务器提供一个连接点（introduction）. 这条消息识别用户并选择文件树去访问.attach 消息会使客户端会拥有一个指向目标文件树根目录的连接，当然这个目录通过 fid 来被识别.Auth 消息包含一个 afid，用来进行认证. 一旦认证协议完成，相同的 afid 会被 attach 提供给用户来保证成功进入并获取服务.
3. stat，wstat：stat 消息获取文件信息.Stat 字段包括文件名，读写、执行权限，访问和修改次数，所有者和组的信息.Wstat 允许默写文件的属性被做某些改动.
4. flush：flush 信息可以终止某个请求. 当服务器收到 Tflush，它就不会回复带有 oldtag 标记的消息并立刻发送 Rflush. 客户端必须等待直到得到 Rflush. 这时 oldtag 会被回收.
5. write，read，open，create，remove 功能都如上面简介介绍.

另外，Styx 对于安全性也做了一定工作，提供了许多安全机制来处理会影响系统整体性和安全性的危险动作. 统一的文件通信协议包含着用户和组的识别码. 比如，在收到一个打开文件请求时，服务器会检查并匹配用户 id. 这个机制和通用操作系统比较类似.Styx 用通过网路连接的文件系统方

式提供远程资源. 这种获取远程资源的方式对应用程序是透明的，所以认证不需要特别提供. 例如，在 inferno 中，客户端和服务器间的通信渠道中会包含许多加密和消息摘要的协议. 与通用文件服务器和一些电话通讯领域一样，所有对于资源的运用都会经过一定的认证，这也保证了远程管理的安全性.

值得一提的是 Styx 协议将请求翻译为必要的字序列并将其在通信通道中传递. 所以 Styx 协议适应于 ISO 标准的 OSI Session Layer Level. 所以它独立于机器结构并成功地应用于不同指令集和数据格式的机器中.

1.1.2 ns - 显示当前命名空间

在立项依据中，我们简要介绍了命名空间的概念，在 Inferno 系统中，命名空间是一系列挂载点（mount points）和绑定（binds）的集合，与 UNIX 系统的挂载点相似. 在 Inferno 系统中，你可以把一个使用 styx 协议的服务器挂载在一个文件描述符上（如一个网络连接，或者一个程序的管道)，因此，挂载在命名空间中引入了一个新的文件树；另一方面，挂载还可以仅仅使命名空间的一部分作为一个别名出现在另一个命名空间中.

命名空间包括两个部分：常规路径，以“/”开始；和一个“特殊”的路径集合，以“#”开始，后面跟单个字符. 这些特殊路径是内核设备，为内核服务的文件树，频繁访问的硬件（如硬盘）或者内核数据结构（如处理器）. 只有命名空间中的常规路径可以通过 bind、mount、unmount 指令修改.ns 命令用来打印当前的命名空间，在 shell 中执行 ns 命令打印出的是 shell 的命名空间. 为了举例说明，下面是在初始 Inferno 下运行 ns 的输出：

```
; ns
bind / /
bind -ac '#U' /
bind /dev /dev
bind -b '#^' /dev
bind /chan /chan
bind -b '#^' /chan
bind -b '#m' /dev
bind -b '#c' /dev
bind '#p' /prog
```

```
bind '#d' /fd
bind /net /net
bind -a '#I' /net
bind -a /dev /dev
bind -a /net /net
bind /net.alt /net.alt
bind -a /net.alt /net.alt
bind -c '#e' /env
cd /
;
```

如上面的输出所示，大多数行在常规文件系统中 bind 一个内核设备，例如 #U（Inferno 根目录的内容）在/，#m（鼠标）在/dev 上，#I（网络栈）在/net 上.-a 和 -b 选项分别表示第一条路径的内容出现在原来第二路径的后面和前面（第二条将会包含第一条路径和它自己的集合）. 如果出现选项 -c，目标路径将允许文件创建. 上面出现的命名空间是被 Inferno 初始化代码安装的，作为一种引导系统的方式给出一个合理的默认命名空间.

挂载在命名空间中引入了一个外部文件树，就像 UNIX 系统 mount 一样，文件树被文件描述符上使用 styx 协议访问. 内核通过翻译 open/read-/write/stat/stc 系统调用成 styx 信息来控制 styx 部分，然后在返回值中回复信息.mount 系统调用要求文件描述符使用 styx 协议通信.mount 程序挂载 3 种类型的 styx 服务器拥有很方便的语法：

- mount /path/to/styx/file target, 通过打开/path/to/styx/file 取得文件描述符
- mount net!www.example.org!styx target, 通过访问 net!www.a.org!styx 取得文件描述符
- mount program target, 开始程序并且用它的标准输入作为文件描述符

ns 显示给定 pid 或者默认它本身的命名空间结构，以/prog/pid/ns 的内容为基础，打印一系列 bind 和 mount 命令，如果它被执行，会重建一个相同的命名空间. 如果任何涉及到的文件因 mount 或 bind 命令被改名，那么就显示文件的原始名字.

1.1.3 bind, mount, unmount - 修改命名空间

bind 和 mount 命令修改当前进程和在一个命名空间组中其他进程的命名空间，目标是一个存在的文件或当前将要修改的命名空间的目录名. 对于 bind，目标是一个已经存在的文件或当前命名空间下的目录名. 在 bind 命令成功执行之后，目标文件名就是原来的东西起的一个别名；如果修改没有隐藏它，目标将依然指向他的原来的文件. 源和目标必须是同一个类型，要么都是目录，要么都是文件. 对于 mount，源可以是一个 shell 命令，一个网址，或者一个文件名，如果源被花括号扩起来，那么它将作为一个 sh 命令的调用，如果源中包含感叹号或者没有文件，他将被作为一个网址.

bind 和 mount 命令的工作可以被 unmount 命令撤消，如果给 unmount 两个参数，他会撤消相同参数的 bind 或 mount 命令，如果只给它一个参数目标上所有 bind 过和 mount 过的东西都会被撤消. 注意，当使用 bind 或 unmount 命令时，内核设备名称中的 # 字符必须被引号引起来否则 shell 会把它看做注释的开头.

修改命名空间的常用选项：

- -b：mount 和 bind 均有效. 把源目录加入被目标目录代表的联合目录开头.
- -a：mount 和 bind 均有效. 把源目录加入被目标目录代表的联合目录结尾.
- -c：这个选项可以加在上面两个选项上，用来允许在联合目录中的创建. 当在联合目录中创建一个新的文件时，它将被放在联合目录中第一个 mount 或 bind 带有 -c 选项的元素中，如果那个目录没有写权限，将会创建失败.
- -q：如果 bind 或 mount 失败，不打印诊断信息，安静退出.
- -A：只对 mount 有效. 在执行 mount 之前不认证服务连接.

1.1.4 os - 寄主操作系统接口（对寄生安装的 Inferno 有效）

os 指令将使得我们使用 C 语言开发的程序在 Inferno 系统中运行.

os 使用使用 cmd 设备在寄主系统执行命令，如果存在 -m 选项，os 会使用在挂载点的设备，否则将会被假设在/cmd 下，如果必要还会被 bind

在本地命名空间下.-d 选项将会导致命令运行在 dir 目录下，如果 dir 目录不存在或者无法访问就会报错，且指令不会执行. 命令的标准输出和标准错误会出现在 os 命令本身的标准输出和标准错误上，os 复制标准输入到远程命令的标准输入；如果命令没输入，就会重定向 os 的输入到/dev/null. 当 cmd 终止后，os 命令就终止了，它的退出状态返回的是 cmd 的退出状态. 如果命令被杀死或退出（比如缺少输入或输出），寄主自己的进程控制办法将试图杀死仍在运行的 cmd，-b（background）选项将制止这个行为.-n 选项导致 cmd 在低于正常优先级下运行.-N 选项把低优先级设置为特殊的等级，从 1 到 3.

1.1.5 cpu - 执行一个远程的命令

cpu 命令向主机拨号（使用 tcp 网络如果网络没有显式给出）. 连接后，向外传输本地的命名空间并执行远程机器上的指令. 本地命名空间对于/n/client 中的命令是可见的；本地的设备文件被 bind 到了远程设备目录下. 如果命令没有给出，那么/dis/sh 就处在运行之中.

-C 选项设定了 authentication 被摘要或者加密的算法，具体的摘要算法有：MD4,MD5 等，具体的加密算法有：RC4,DES,CBC,ECB 等算法. 默认的是不用任何算法.

1.1.6 通信模式 - 主从模式

之前我们调研了其他并行框架如 MPI, CUDA, OpenMP 等背景与实现方式. 纵观 MPI, 它的通信模式主要有 4 种:

- 标准通信模式: 对等模式与主从模式.
- 缓存通信模式
- 同步通信模式
- 就绪通信模式

经过深入了解, 我们决定模仿 MPI 的主从模式去实现我们的并行计算框架. 因为基于 Inferno 强大的分布式特点, 我们可以简单的用 bind 的命令将远程的 Inferno 节点的资源挂载在本地, 亦或是用 cpu 命令直接在远程端运行给定程序. 所以我们可以选择一台节点当主节点, 然后将其余从节点全

部挂载在主节点上. 由主节点分发任务, 响应从节点请求, 进行从节点间调度等等.

基于主从模式的通讯方式在 Inferno 下开发并行框架可以大大加快我

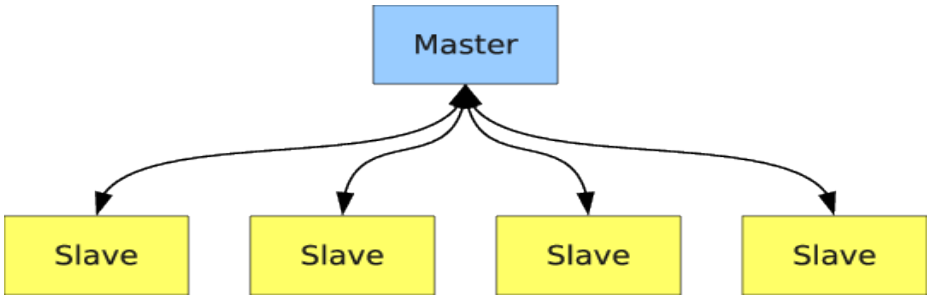

Figure 1.1: 主从模式

们推进速度.

1.1.7 file2chan - 节点间通讯

因为我们要实现并行计算框架, 一个亟待解决的问题就是如何做节点间通讯. 最初我们的想法是直接用一个文件来做共享池. 然而后来在此基础上进行实现时发现困难重重:

1. 直接用文件做共享难以解决同步的问题. 因为直接用文件做共享池, 这样在运行时没法动态的加锁, 解锁, 不方便解决同步问题.
2. 难以实现主节点的动态任务分配.
3. 难以实现主节点对从节点请求的动态响应.
4. 光是一个文件难以做到传输各种格式不同的通讯信息.
5. 直接在磁盘中暴露中间计算内容, 缺少加密.

经过我们重重调研, 发现 Inferno 提供了一个叫做 file2chan 的脚本工具.

file2chan 是一个 sh 的可加载模块，可以在命名空间中创建一个可以由 shell 脚本决定其性质的文件.file2chan 在命名空间中创建一个文件名，同时产生一个新的线程来为这个文件服务，如果成功了，环境变量 $apid 就会被

设置成新线程的 pid，否则返回一个错误状态（non-nil）.readcmd、writecmd 和 closecmd 都应该是可执行的指令块. 之后，每当一个进程从文件名读入，就会调用 readcmd；每当一个进程对文件名写入，就会调用 writecmd；每当一个从文件名打开的文件关闭，就会调用 closecmd.

如果我们采用 file2chan, 则可以轻松的解决上述问题. 具体设想与实现可以参考概要设计报告部分.

1.2 创新点

1.2.1 实现 MapReduce 计算模型

尚未有人、或极少人在 Inferno 上做过并行计算. 然而，Inferno 系统本身的特性使得它能很方便的实现并行计算. 为了进一步完善我们小组的并行计算模型，我们也在 Inferno 上实现了 MapReduce 这一编程模型. 通过 MapReduce, 我们可以把一个大作业拆分成多个小作业，而用户只需要决定拆分成多少份，以及定义作业本身. 因此，不熟悉并行编程的程序员也能充分发挥分布式系统的威力. 事实证明，Inferno 在实现并行计算框架这一点上较其它系统有天然的优势，只要进行少量的工作就完成了 MapReduce 在 Inferno 上的移植.

1.2.2 多节点视为整体的一个节点

集群是一组相互独立的、通过高速网络互联的计算机，它们构成了一个组，并以单一系统的模式加以管理. 如今主流的集群实现方式中，集群中节点间的数据交换通过数据共享空间或者点对点的通信实现，各个节点事实上并不能真正成为一个整体，例如基于 MPI(Message Passing Interface) 的并行框架. 而 inferno 通过导入 Namespace（命名空间），使所有的资源都以分层次的文件的形式存在于文件系统中，因此应用程序可以以同样的方式访问远程的节点和本地节点. 故而虽然这个系统上包含了很多的节点, 但是在用户和应用程序看来, 这就像是一个节点一样.

1.2.3 Inferno 在 64 位平台上的移植

最新的 inferno 第四版发行于 2004 年，支持的平台有 ARM,PA-RISC,MIPS,PowerOC,SPARC,X86. 但如今市面上的平台基本上都是 64 位的了，因此要想在 64 位的平台上运行 inferno，就要进行移植. 为了让

Inferno 能成功地在市面上绝大多数电脑上运行，把更多的平台纳入我们的并行框架中，我们修改了 Inferno 的部分代码，将其移植到了 64 位的平台上.

1.2.4 以外部解释器的形式实现并行计算

之前调研了其他著名并行计算框架的使用方式:

- CUDA, MPI : 用户根据具体的 MPI 实现框架 (Intel MPI, OpenMPI 等) 给定的 API 接口, 调用具体的库函数, 进行串行代码的并行化移植.
- OpenMP : 在要并行的代码块前加预处理指令 #pragma. 编译器编译时则自动会链接 OpenMP 库, 编译出并行化代码.

我们借鉴了 OpenMP 的并行化的技术, 提出了使用外部解释器思想. 我们是针对 Inferno 的 shell 实现并行计算框架, 而 shell 并不需要编译, 是通过 Inferno 中自带的 sh 逐行解释执行的. 然而直接对 Inferno 中的 sh 动手脚是不可能的,sh 部分代码量高达好几万行, 没有注释并且还使用了部分内核接口. 我们之前调研发现 Inferno 的 shell 本身就自带 bind, cpu, file2chan 等工具, 可以直接编写能在 Inferno 的 shell 上运行的并行化代码. 所以我们决定实现一个外部解释器, 将用户的串行代码按照给定的需求解释成能直接在 Inferno 的 shell 中直接运行的并行化代码.

这点和 OpenMP 又有极大的不同,OpenMP 并行化的程序在运行时还需要调用 OpenMP 库. 而我们则要实现串行代码再经过解释器解释之后可以直接在 Inferno 的 sh 上运行, 不需要再调用其他库, 可以说是得到了在 Inferno 平台下“完全并行化”的代码.

1.2.5 以 C 语言而非 Limbo 语言来编写解释器程序

在 Inferno 里，我们需要用 Limbo 来写应用程序.Limbo 是一种用于分布式系统的编程语言. 然而 Limbo 是一门不流行的语言，相关文档较少，因此学习这门语言势必需要花费较多的时间与精力. 但是，假如我们是以应用程序的形式在别的系统上运行 Inferno，那么我们就可以通过 os 这条命令，来间接地在 Inferno 上运行 host 机器上的程序. 于是，我们小组选择了用 C

语言来写解释器，通过 os 命令来间接地让解释器在 Inferno 上运行，大大加快了开发进度.

1.2.6 对 file2chan 的读写方法封装, 实现一个文件完成各种类型通讯

此部分具体见概要设计报告中的 sharedPool - 节点间通讯一节.

1.3 概要设计报告

1.3.1 总体路线

在理论依据部分和创新点部分已经提及了我们的部分设计思路. 我们的最终目的就是实现一个外部解释器, 可以根据需求将用户给定的串行程序翻译成基于主从模式的, 可以在 Inferno 的 shell 环境下直接运行的并行化代码.

总的开发阶段可以划分为如下两个部分:

- 自己先摸索出在 Inferno 中将串行程序并行化改写的方法, 并对具体语句总结出一套模式化改写规则 (如对一段 for 改如何改写)
- 开发解释器, 按照之前总结的改写规则将串行代码翻译为并行代码.

对于第一个阶段我们已经完成部分, 具体成果如下所示.

1.3.2 一个串行程序

```
#!/dis/sh -n

load std mpexpr

subfn fac {
    ans = 1
    for j in ${expr 1 $1 seq} {
        ans = ${expr $ans $j '*'}
    }
    result = $ans
```

```
    if {~ $1 0} {
       result = 1
    }
}

subfn c {
      n = $1
      r = $2
      n_fac = ${fac $n}
      r_fac = ${fac $r}
      n_r_fac = ${fac ${expr $n $r '-'}}
      ans = ${expr $r_fac $n_r_fac '*'}
      result = ${expr $n_fac $ans '/'}
}

subfn calc {
      n = $1
      num = 0
      for r in ${expr 1 $n seq} {
          if {~ ${expr ${c $n $r} 1000000 '>'}
             1} {
         num = ${expr $num 1 '+'}
      }
      }
      result = $num
}

ans = 0
for i in ${expr 23 100 seq} {
      ans = ${expr $ans ${calc $i} '+'}
}
echo $ans
```

先简要的解释一下这个程序.subfn fac 实现了一个计算阶乘的子函数.subfn c 实现了一个组合数的子函数.subfn calc 实现了一个统计函数, 对于给定的 n 可以统计有多少个 $r \leq n$ 满足 $C(n,r) > 1000000$.

最外层则是一个 i 从 23 到 100 变化的 for 循环, 将所有的 calc(i) 相加输出. 值得注意的是在 Inferno 下任何算数运算都采用后缀表达式并通过调用内嵌子函数 expr 完成.

本身这个程序没有任何意义, 举这个程序为例子只是为了演示如何将一段串行代码并行化.

1.3.3 改写的主从式并行代码

master 部分

```
#!/dis/sh -n

load std mpexpr file2chan

host = $*
sharedPath = /n/client/tmp/sharedPool
answer = 0
masterLogPath = /pool/master.log

upper = 100
lower = 23
lock = 0

readmodes = get

subfn cannotWrite {
    result = 0
    if {~ $lock 1} {result = 1}
    for readmode in $readmodes {
        if {~ $readmode $mode} {result = 1}
    }
```

```
}

file2chan /tmp/sharedPool {
    while {~ $lock 1} {}
    lock = 1
    if {~ $mode 'get'} {
        if {~ ${expr $lower $upper '>'} 1} {
        echo done | putrdata
    } {
            echo $lower | putrdata
        lower = ${expr $lower 1 '+'}
    }
    }
    mode = ''
    lock = 0
} {
    if {~ ${cannotWrite} 1} {} {
        lock = 1
        (client_pid mode writeData) = fetchwdata
        if {~ $mode 'put'} {
            answer = ${expr $answer $writeData
                '+'}
            echo from $client_pid current
                answer is $answer >>
                $masterLogPath
        }
    touch /pool/$client_pid
        lock = 0
    }
}

fn checkHost {
    if {~ $#host 0} {
```

```
        echo please give me the hostname >>
            $masterLogPath
        raise args
    }
}

fn readyHost {
    for hostname in $host {
        cpu $hostname sh /n/client/parallel/
            worker.sh $sharedPath&
    }
}

checkHost
readyHost
```

worker 部分

```
#!/dis/sh -n

load std mpexpr

subfn fac {
    ans = 1
    for j in ${expr 1 $1 seq} {
        ans = ${expr $ans $j '*'}
    }
    result = $ans
    if {~ $1 0} {
       result = 1
    }
}

subfn c {
```

```
        n = $1
        r = $2
        n_fac = ${fac $n}
        r_fac = ${fac $r}
        n_r_fac = ${fac ${expr $n $r '-'}}
        ans = ${expr $r_fac $n_r_fac '*'}
        result = ${expr $n_fac $ans '/'}
}

subfn calc {
        n = $1
        num = 0
        for r in ${expr 1 $n seq} {
            if {~ ${expr ${c $n $r} 1000000 '>'}
                1} {
               num = ${expr $num 1 '+'}
        }
        }
        result = $num
}

subfn writeRequest {
while {! ftest -e '/n/client/pool/'^${pid}} {
              echo ${pid} $* > $sharedPath
        }
        rm /n/client/pool/${pid}
}

subfn getTask {
        ${writeRequest 'get'}
        result = {sed 1q < $sharedPath}
}
```

```
answer = 0
sharedPath = $1
hostname = os hostname
workerLogPath = /n/client/pool/$hostname.log

echo $hostname is pending >> $workerLogPath

while {} {
        current = ${getTask}
        if {~ $current 'done'} {raise break}
        echo $hostname fetched $current >>
            $workerLogPath
        current_ans = ${calc $current}
        ${writeRequest 'put' $current_ans}
}

echo $hostname is finished >> $workerLogPath
```

1.3.4 sharedPool - 节点间通讯

首先在 master 部分我们通过 file2chan 定义了一个虚拟文件/tmp/sharedPool 用作节点间通讯. 然而由于节点间通讯消息种类十分繁杂 (例如主节点给从节点分配任务, 从节点计算完毕将结果提交给主节点), 各自的格式也都不一样. 如果开多个 file2chan 则必定会带来管理的紊乱. 于是我们创新性的采用封装了 file2chan 的 read,write 方法使得一个 file2chan 可以完成各种类型的信息通讯.

当想使用 sharedPool 进行消息通讯时, 首先向 sharedPool 写入具体的操作类型.sharedPool 中的我写好的 read 的方法会读取给定的操作类型, 并在内部根据给定操作类型切换响应的模式来应对.

例如在我这个例子中, 当 worker 想要向 master 索求数据时先向 sharedPool 写入 get, 然后再对 sharedPool 执行读取操作即可读到主节点分配给它数据. 当 worker 计算完成, 要把结果反馈给 master 时就先向 sharedPool 写入 put, 然后再往 sharedPool 写入算得的数据, 此时就会触发 sharedPool

的 write 方法, 自动将本次 worker 向 master 提交的结果计入总结果.

1.3.5 节点间通讯同步

Inferno 本身已经对 file2chan 做了同步处理, 当你往里写时就不能读, 读时就不能写. 一个节点在读的同时其他节点就不能读, 一个节点在写时其他节点就不能写. 我们唯一要做的是保持事务的原子性. 因为我们之前封装了 file2chan 的 read,write 方法使得其支持各种类型的通讯信息, 这带来方便也带来了弊端. 例如一个节点往 sharedPool 中写入了 get, 于是 sharedPool 内部切换为 get 模式等待这个节点来取数据. 然而若在此时另外一个节点计算完了数据想向 sharedPool 提交数据, 于是在之前那个节点读数据之前又往 sharedPool 写入了 put, 使得其内部转换为了 put 模式. 那么之前那个节点再读数据就会发生错误. 我们要做的就是使得一个节点向 sharedPool 指明操作类型之后的下一步只接受这个节点的请求 (即事务的原子性).

我们的实现方法是当 sharedPool 接收到 a 节点指定的命令类型之后, 马上锁定 a 节点. 下一步只接受来自 a 节点的请求. 如果 b 节点此时再往 sharedPool 写入就会无效. 然而这还有一问题就是要让 b 节点等待一会再执行写入操作, 而不是直接跳过这次写入操作, 否则会导致信息丢失. 如果 file2chan 可以提供 write 方法的返回值, 通过告诉节点 b 执行 write 方法的成功与否来让他继续执行或等待.

可惜的是 Inferno 中没有提供对 file2chan 中 read,write 方法返回值的支持. 于是我们创新性的采用文件名的方式来解决这个问题. 由于每个节点都对应一个进程号 pid, 当 pid 为 x 的节点写操作成功时就在/tmp 目录下生成一个名为 x 的文件. 当 pid 为 x 的进程检测到目录中有 x 这个文件时就知道了它本次 write 操作是成功的, 可以删掉这个文件 (消除痕迹) 继续往下执行, 否则等待. 通过方式我们绕开了 Inferno 中 file2chan 的限制, 成功的实现了同步效果.

1.3.6 执行效果

为了便于调试, 我们将 master 和 worker 的屏幕输出信息都重定向到了日志文件. 执行环境是我和刘旭彤的电脑 (通过 wifi 连接在同个局域网下), 我的机子充当主节点, 刘旭彤的机子充当从节点. 在我的 Inferno 中执行 cpu master.sh 192.168.1.1(局域网中我的 IP) 192.168.1.10(局域网中刘旭

形的 IP) 便可实现两个节点对跑.

运行完毕后/pool 目录中有 3 个日志文件.

liuxt-MacBookAir-Invalid-entry-length-DMI-table-is-broken-Stop.log

```
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop is pending
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 24
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 28
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 32
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 37
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 41
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 45
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 49
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 52
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 56
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 59
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 62
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 66
liuxt-MacBookAir-Invalid-entry-length-DMI-table
```

```
    -is-broken-Stop fetched 69
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 72
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 75
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 79
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 82
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 85
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 88
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 92
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 95
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop fetched 98
liuxt-MacBookAir-Invalid-entry-length-DMI-table
    -is-broken-Stop is finished
```

macbook.log

```
macbook is pending
macbook fetched 23
macbook fetched 25
macbook fetched 26
macbook fetched 27
macbook fetched 29
macbook fetched 30
macbook fetched 31
macbook fetched 33
macbook fetched 34
```

```
macbook fetched 35
macbook fetched 36
macbook fetched 38
macbook fetched 39
macbook fetched 40
macbook fetched 42
macbook fetched 43
macbook fetched 44
macbook fetched 46
macbook fetched 47
macbook fetched 48
macbook fetched 50
macbook fetched 51
macbook fetched 53
macbook fetched 54
macbook fetched 55
macbook fetched 57
macbook fetched 58
macbook fetched 60
macbook fetched 61
macbook fetched 63
macbook fetched 64
macbook fetched 65
macbook fetched 67
macbook fetched 68
macbook fetched 70
macbook fetched 71
macbook fetched 73
macbook fetched 74
macbook fetched 76
macbook fetched 77
macbook fetched 78
macbook fetched 80
```

```
macbook fetched 81
macbook fetched 83
macbook fetched 84
macbook fetched 86
macbook fetched 87
macbook fetched 89
macbook fetched 90
macbook fetched 91
macbook fetched 93
macbook fetched 94
macbook fetched 96
macbook fetched 97
macbook fetched 99
macbook fetched 100
macbook is finished
```

master.log

```
from 77 current answer is 4
from 77 current answer is 14
from 893 current answer is 21
from 77 current answer is 32
from 77 current answer is 44
from 77 current answer is 60
from 77 current answer is 77
from 893 current answer is 92
from 77 current answer is 110
from 77 current answer is 132
from 77 current answer is 155
from 77 current answer is 179
from 893 current answer is 198
from 77 current answer is 223
from 77 current answer is 250
from 893 current answer is 276
```

```
from 77 current answer is 304
from 77 current answer is 333
from 77 current answer is 364
from 77 current answer is 396
from 893 current answer is 426
from 77 current answer is 461
from 77 current answer is 498
from 77 current answer is 536
from 893 current answer is 572
from 77 current answer is 611
from 77 current answer is 652
from 893 current answer is 692
from 77 current answer is 734
from 77 current answer is 778
from 77 current answer is 823
from 893 current answer is 866
from 77 current answer is 912
from 77 current answer is 960
from 893 current answer is 1007
from 77 current answer is 1056
from 77 current answer is 1107
from 893 current answer is 1157
from 77 current answer is 1209
from 77 current answer is 1263
from 77 current answer is 1318
from 893 current answer is 1371
from 77 current answer is 1427
from 77 current answer is 1485
from 893 current answer is 1542
from 77 current answer is 1601
from 77 current answer is 1662
from 893 current answer is 1722
from 77 current answer is 1784
```

```
from 77 current answer is 1850
from 893 current answer is 1915
from 77 current answer is 1982
from 77 current answer is 2051
from 893 current answer is 2119
from 77 current answer is 2189
from 77 current answer is 2260
from 77 current answer is 2333
from 893 current answer is 2405
from 77 current answer is 2479
from 77 current answer is 2555
from 893 current answer is 2630
from 77 current answer is 2707
from 77 current answer is 2786
from 893 current answer is 2864
from 77 current answer is 2944
from 77 current answer is 3026
from 893 current answer is 3107
from 77 current answer is 3190
from 77 current answer is 3274
from 77 current answer is 3360
from 893 current answer is 3445
from 77 current answer is 3532
from 77 current answer is 3621
from 893 current answer is 3709
from 77 current answer is 3799
from 77 current answer is 3891
from 77 current answer is 3984
from 893 current answer is 4075
```

可以看到我的机子 (macbook,pid77) 和刘旭彤的机子 (liuxt-MacBookAir,pid893) 并行的完成了整个任务的计算.

同时我还比较了串行程序运行时间与两个节点并行化运行时间. 对于这

个例子, 串行时间为 1 分 32 秒, 两个节点并行化运行时间为 56 秒, 可以看到加速比为 1.6(小于 2 是因为节点间负载均衡问题以及节点间通讯调度时间) 效果明显!

1.3.7 Future Work

由于任务调度是直接基于抢占式的, 由于网络关系 (我在这个例子中既充当了主节点又充当了从节点, 作为从节点向主节点抢占任务的速度肯定快于刘旭彤) 可以看到节点间负载并不均衡. 所以我们下一步要做的首要工作是进行节点间负载均衡.

其次我们将继续摸索串行程序并行化的改写方法. 当积累到一定程度时, 并可开始解释器的编写工作.