

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 调研报告	5
1.1 项目背景及实践意义	5
1.1.1 项目的背景	5
1.1.2 分布式操作系统	6
1.1.3 分布式计算	6
1.1.4 分布式系统实现并行计算的主要问题	7
1.1.5 Inferno 操作系统	8
1.2 小组成员相关背景	9
1.3 立项依据	10
1.3.1 项目意义	10
1.3.2 理论依据	10
1.4 目前所调研到的相关工作	12
1.4.1 Inferno 相关调研	12
1.4.2 常见并行框架概况	12
1.5 OpenCL 调研	14
1.5.1 简介	14
1.5.2 主机设备交互	15
1.5.3 OpenCL 架构	15
1.6 MPI 调研	20
1.6.1 MPI 数据类型	20
1.6.2 通信域	21
1.6.3 MPI 调用接口	21
1.6.4 MPI 并行编程模式	22
1.7 想法来源及可能遇到的挑战	24
2 可行性研究报告	25
2.1 理论依据与技术依据	25
2.1.1 Sytx 协议及其具体实现	25
2.1.2 ns - 显示当前命名空间	28
2.1.3 bind, mount, unmount - 修改命名空间	29
2.1.4 os - 寄主操作系统接口（对寄生安装的 Inferno 有效）	30
2.1.5 cpu - 执行一个远程的命令	31

2.1.6	file2chan - 节点间通讯	31
2.2	创新点	32
2.2.1	良好的异构支持	32
2.2.2	实现 MapReduce 计算模型	32
2.2.3	多节点视为整体的一个节点	32
2.2.4	Inferno 在 64 位平台上的移植	33
2.2.5	以外部解释器的形式实现并行计算	33
2.2.6	以 C 语言而非 Limbo 语言来编写解释器程序	34
2.2.7	对 file2chan 的读写方法封装, 实现一个文件完成各种 类型通讯	34
3	概要设计报告	34
3.1	总体路线	34
3.2	平台模型 - OpenCL 的 host & device 模式	35
3.3	异构性	35
3.3.1	抢占式任务分配	35
3.3.2	细并行粒度	36
3.4	Raspberry Pi2	36
4	理论与实践	37
4.1	Inferno 在 Raspberry Pi 移植	37
4.2	初步尝试 — 一个串程序	38
4.3	初步尝试 - 改写的 host & device 式并行代码	40
4.4	file2chan - sharedPool 的基础	44
4.5	sharedPool - 节点间通讯 (任务分配收发池)	61
4.6	device 部分代码	61
4.7	host 部分代码	61
4.8	并行框架开放的三个函数	62
4.8.1	map	62
4.8.2	reduce	63
4.8.3	merge	65
4.9	函数并行化原理与容错机制	66
4.9.1	map	66
4.9.2	reduce	67

4.9.3	merge	68
4.10	并行化执行流程图	68
4.11	解释器的总体架构	73
5	并行计算框架的安装与用法	74
5.1	安装方法	74
5.2	配置 device_ip	74
5.3	user.sh 的编写	74
5.4	执行一个并行化程序	77
6	成果与不足	78
6.1	向官方提交了 2 个 bug 与 1 个 patch	78
6.1.1	arm 架构下的编译 bug	78
6.1.2	arm 架构下的网络接口 bug	78
6.1.3	有关 x86 编译问题的 patch	78
6.2	日志功能方便调试	80
6.3	故障修复	82
6.4	跨平台性	82
6.5	高度可扩展性与可配置性	83
6.6	benchmark	84
6.6.1	各个设备的配置	84
6.6.2	benchmark 内容	85
6.6.3	性能基准测试结果	89
6.6.4	使用并行计算框架进行并行 benchmark 测试	90
6.7	与其他并行计算框架的对比	91
6.7.1	异构性	91
6.7.2	部署简单	92
6.7.3	编程简单	93
6.7.4	支持故障恢复	102
6.8	不足点	103
6.9	完成过程中遇到的困难	104
7	致谢	104
8	参考文献	105

1 调研报告

1.1 项目背景及实践意义

1.1.1 项目的背景

如今人类各个学科繁多、涉及面广、分类细致，而且不同学科之间又有交叉。而如今很多的学科都需要进行大量的计算，如天文学研究组织需要计算机来分析太空脉冲，星位移动；生物学家需要计算机来模拟蛋白质的折叠过程；药物学家需要计算机来研制克服爱滋病或非典的药物；数学家需要计算机来计算最大的质数和圆周率的更精确值；经济学家也要用计算机分析计算在几万种因素考虑下某个企业/城市/国家的发展方向从而宏观调控。由此可见，未来科学的发展离不开计算。在很多时候，为了进行研究而特地去购买一台专用的超级计算机往往是不现实的。而分布式计算，因为其高效而便宜的特点，越来越受社会的关注。

为了说明目前国内分布式计算的现状，我们参考了 BOINC 平台的一些数据。

BOINC，也即伯克利开放式网络计算平台（Berkeley Open Infrastructure for Network Computing），是目前主流的分布式计算平台之一。作为一个高性能的分布式计算平台，截止至 2015 年 1 月 16 日，BOINC 在全球有 235,980 位活跃的参与者以及 692,208 台活跃的主机，提供约 9.871 petaFLOPS 的运算能力（天河二号超级计算机运算速度约为 33.86 petaFLOPS）。

以生命科学为例，下面是一些运行在 BOINC 平台下的项目：

1. Discovering Dengue Drugs：针对登革热、丙型肝炎、西尼罗河病毒和黄热病病毒发现有前途的药物先导化合物
2. Genome Comparison：染色体对比研究
3. Help Conquer Cancer：帮助科学家征服癌症
4. Help Defeat Cancer：帮助科学家对抗癌症
5. Human Proteome Folding：人类蛋白质折叠研究
6. Human Proteome Folding 2：人类蛋白质折叠第 2 阶段研究

- 7. Help Cure Muscular Dystrophy : 针对肌肉营养失调和其他神经肌肉型疾病的研究
- 8. Nutritious Rice for the World : 针对水稻预测蛋白质结构的研究, 以提高水稻产量

1.1.2 分布式操作系统

分布式操作系统 (Distributed operating system) 是在互不依赖的、联网的、相互通信的、物理上分离的多个节点上的软件. 每个独立的节点包含一部分整体的操作系统上软件的子集. 每个子集都可分为两个不同的部分. 第一部分是一个通用的微内核, 这个内核直接控制节点上的硬件资源. 第二部分是一些高层次的系统管理程序, 用来协调节点上独立的和相互协作的活动. 这两个部分可以把多节点上的资源和处理能力整合成一个高效并且稳定的系统. 因此, 尽管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.

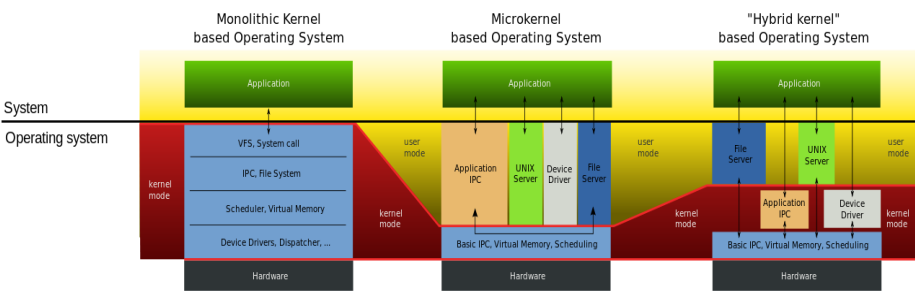


Figure 1.1: 分布式操作系统的架构

1.1.3 分布式计算

分布式计算是指把一个需要拥有强大计算能力的超级计算机才能解决的问题分割成小部分, 通过分配给许多计算机单独进行计算, 整合汇总的局部结果最终得到最终结果, 达到虚拟计算机解决大型问题的效果.

如今很多领域都采用了这种用分布式系统搭建集群并实现并行计算的方式来进行复杂问题的处理. 例如蛋白质疾病分布式计算项目. 这个项目主要研究蛋白质折叠、误折、聚合及由此过程引起的一些相关疾病的分布式计算工程. 该项目使用联网式的计算方式和大量的分布式计算能力来模拟蛋

白质折叠的过程, 该过程需要尽量大量、复杂的计算. 通过利用世界各地闲散的计算资源, 我们可以加快该项目的进程.

1.1.4 分布式系统实现并行计算的主要问题

1. 分布式的数据和文件管理: 在搭建起大规模的集群时, 必须考虑数据分布存储的管理以及访问问题. 唯有解决了这个问题, 才能够保证并行计算结果的正确性以及并行计算的性能.
2. 数据访问和通信控制: 在分布式存储访问结构的系统中, 数据在不同节点间传输, 由于不同节点的运算速度不一致, 此外, 还有数据访问 / 通信的时间延迟等问题的存在, 所以必须考虑数据和计算同步问题.
3. 可靠性: 在某些情况下, 如网络连接失败, 节点可能会失效, 进而导致数据丢失、程序终止、乃至系统崩溃等问题的发生. 因此, 为保证可靠性, 必须考虑数据丢失后的恢复以及程序和系统崩溃后的恢复.
4. 网络互联结构的选择: 不同的节点链接的结构会对系统整体的可靠性以及整体的运算速度产生重大影响. 因此, 必须根据实际需要、实际情况, 权衡各个因素, 选择最合适的节点链接结构, 如主从非对称结构、环形结构、互联网络结构等.
5. 并行计算的任务划分和算法设计: 并行计算需要将一个大的计算任务分解成数量合适的小任务, 把这些小任务分配给不同的节点或者处理器进行处理, 最终收集各个节点 / 处理器返回的局部结果, 并最终整合为一个大的结果. 其中并行计算的形式主要有算法分解和数据划分. 在分解和划分时, 要充分利用各节点、处理器的性能, 从而提高运算速度.
6. 并行计算程序设计框架的设计以及实现: 为了保证并行计算的正确性以及高效性, 程序员需要考虑各种各样繁琐的技术细节: 数据的存储管理、计算任务的划分、任务的调度执行、数据和计算的同步、结果的收集、节点失效后的恢复等. 如何使这些底层细节透明化、自动实现并行计算, 是一个很重要的问题.
7. 如何度量并行计算的性能: 度量并行计算性能的指标是并行计算技术的核心. 唯有有了一个确定的指标, 才能设计出合适的算法以及实现方式.

1.1.5 Inferno 操作系统

根据调研, 我们最终决定使用开源的分布式操作系统 Inferno 来实现分布式计算.

Inferno 是一个由贝尔实验室研究发明的开源的分布式操作系统, 现在由 Vita Nuova 继续发展和维护. 它基于贝尔实验室 plan9 的经验和该实验室对 os、语言、实时编译器、网络、移植等相关方面的研究.

该系统定义了一个名为 Dis 的虚拟机,Dis 可在不同的机器上实现.

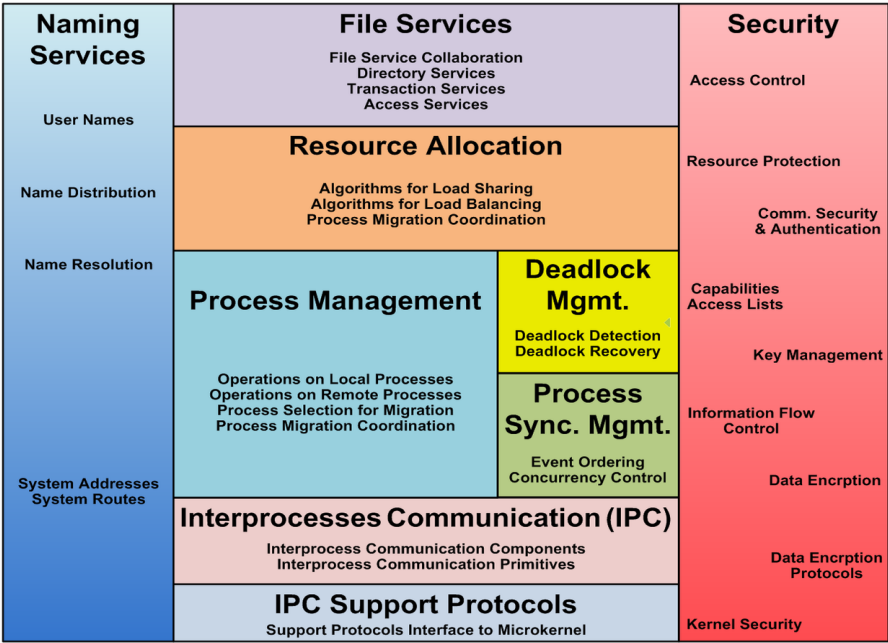


Figure 1.2: Inferno 的架构

Inferno 也提供了一个虚拟的、提供相同借口的操作系统, 这个操作系统可以让用户在硬件上直接运行 Inferno 或者在其它系统中以应用程序的方式运行 Inferno.

为了让应用程序以统一的方式进行文件操作, 如打开、关闭、读写, Inferno 采用了一种名为 Styx 的通信协议.

作为一个分布式操作系统, Inferno 的设计基于以下三个基本原则:

- Resources as files: 所有的资源都以分层次的文件的形式存在于文件系统

统中.

- Namespaces: 在应用程序看来, 网络是一个单一的、条理清楚的 namespace (命名空间). 虽然这个 namespace 看起来像是一个分层次的文件系统, 但实际上, 它表示的是分散的资源.
- Standard communication: 通过一个名为 Styx 的协议来访问所有本地和远程的资源.

1.2 小组成员相关背景

阮震元

1. 第二届全国 RDMA 编程挑战赛冠军
2. 现任科大超算曙光队主力队员, 7 月将代表科大出征国际超算竞赛
3. 高中时获全国信息学联赛一等奖
4. Topcoder SRM DIV 1
5. Coderforces DIV 1
6. 中国最大计算器论坛 cnCalc 超级版主

5 年编程开发经验, 熟悉 C, C++, Java, Python, Scala, Pascal, Deiphi 等语言, CUDA, OPENMP, MPI 等并行技术. 有 JavaSE, JavaWeb 开发经历, Linux 运维经验. 高中时获信息学联赛一等奖, Topcoder SRM DIV 1, Coderforces DIV 1, 有丰富的算法竞赛经历. 大二参加全国 RDMA 编程挑战赛, 具体内容是把分布式处理框架 Spark¹ 的 IO 部分从 TCP/IP 协议移植到 RDMA 协议, 大幅度提高 IO 吞吐率从而达到加速效果, 获冠军. 现任科大超算队队员, 即将出征今年 7 月在德国汉堡举行的国际大学生超算竞赛, 在队内负责大型分子动力学模拟软件 LAMMPS² 的优化加速工作. 同时在队内担任系统管理员一职, 现管理两个超算系统共十余个节点.

¹<http://spark.apache.org/>

²<http://lammps.sandia.gov/>

1.3 立项依据

1.3.1 项目意义

本项目的目标是在具有强大可移植性的分布式操作系统 Inferno 上实现一个通用并行计算框架,使用户在不了解 Inferno 分布式资源细节的情况下方便地开发并行程序.

目前流行的并行计算框架如: 基于消息传递的“裸”并行编程模型 MPI; 使用 Nvidia 加速卡的 CUDA; 基于同步数据流的 MapReduce, Spark, Prlter; 基于数据流水线的 Dryad/DryadLINQ, SCOPE, 基于存储共享的 Piccolo, 用于图计算的 Pregel, Apache 等. 本项目所实现的基于分布式操作系统 Inferno 的分布式结构和 rstyx 网络协议的并行计算框架属于独创性工作.

本项目利用 Inferno 系统方便的分布式资源管理解决了分布式数据存储管理的问题, 通过 rstys 协议解决通信控制问题.

相比 CUDA 而言, 我们并不需要特殊的硬件, 只需每台节点上装有 Inferno. 对于 OpenMP 而言, 我们不仅可以和他一样方便使用一条预处理指令即可自动对 for, while 等实现并行化, 还可以直接看到由我们的解释器翻译出来的并行化代码 (保证可读性和可扩展性), 甚至可以自己再进一步修改优化. 相比 MPI 我们则不需要实现复杂的通信接口, 一切都基于 Inferno 强大的网络协议 Styx, Rstyx. 相比 MapReduce, Spark 而言, 我们利用 Inferno 天然的分布式特点, 使用脚本即可实现类似于 Map, Reduce 的函数.

1.3.2 理论依据

资源即文件 所有 Inferno 的本地和远程资源都被表示为分层文件系统内的一组动态的文件, 这些文件可以表示存储设备、进程、服务、网络和网络连接. 应用程序可以通过使用标准的文件操作要求来操作相关的文件进而访问每个资源. 使用文件为系统中心概念的优点是:

- 文件系统具有跨多种操作系统的简单和易于理解的界面, 文件界面一般有一套定义好的操作, 例如打开, 读, 写.
- 通过文件系统降低了代码量, 并保持 Inferno 系统简单、可靠和高度轻便的特性.
- 对众所周知的文件使用惯例命名, 便于统一和理解.

- 便于确定文件的访问权限和文件许可,可以用于确保多级别的安全性.

文件名和这些文件的动态的内容可以基于每个需求和每个客户端的基础上产生. 例如对于一个传感器资源的数据文件在不同的时间读取就会返回不同的输出, 或者每次读取都会在新的一行添加一个数据. 这种特性是实现并行计算数据共享的基础之一.

命名空间 Inferno 的第二个关键原理就是可计算的命名空间, 通过命名空间每个应用程序会对它需要访问的资源建立一个独一无二的私有视角, 所有的资源被表示为文件的层次结构, 并且通过标准文件访问操作来访问, 一个进程所使用的各种文件和服务都被一个单根分层文件系统组合起来, 叫做命名空间. 一个名称空间内访问的资源可以位于单个客户端或在整个网络中的多个服务器.

命名空间系统的主要优点是应用程序可以使用的资源完全透明, 在应用程序可见的命名空间中挂载的所有动态资源文件, 无论是本地的还是远程的, 该应用程序都可以方便的访问. 例如, 一个图形化的调试器通过读取在/prog 目录下的动态文件来访问关于目前系统进程的信息, 如果用户希望在另一台远程机器上调试, 只需将原来那台机器的/prog 目录挂入目前这台机器, 对于调试器而言, 它只是读取在/prog 目录中的文件, 但并不知道他们来自哪里.

使用标准通信协议来访问资源 Inferno 使用一个被 Inferno 内核或应用程序执行的协议来表示或访问资源. 因为所有的资源都被表示为文件, 包括网络和网络连接, 所以需要一个协议来和提供本地与远程资源的通信. 这个协议是一个叫 9P 的文件服务协议 (Styx 是一个更早的变体). 这种方法就为我们利用已知的技术建立添加远程文件系统的分布式系统提供了一个自然的途径. 拥有一个标准通信协议还提供了一个关注安全的点, Inferno 提供了下面几种安全通信机制:

- 基于证书的用户认证.
- 消息加密.

1.4 目前所调研到的相关工作

1.4.1 Inferno 相关调研

对于 Inferno 系统的调研的目的是利用 Inferno 系统方便的分布式资源管理以及 sytx 通信协议来处理集群搭建以及并行框架问题. 从目前的网络资源来看, Inferno 研发的相关工作主要集中于 Inferno 的开发团队.

我们小组对于 Inferno 的理解主要来源于 Inferno 系统自带的 documentation(doc 目录) 以及 Inferno 官方 documentation 和 tutorial³. Bell 实验室公开的 Documentation⁴.

Inferno 提供了两种模式 native 和 hosted. 其中 hosted 模式是一种应用层虚拟化, 可以让 Inferno 运行在 Linux, Win, OSX 等系统中.

经过我们的调研, Inferno 系统的内核有 80 余万行代码, 并且无法取得到其开发日志. 另外, Inferno 系统已于 2010 年停止维护. 所以我们决定只以 Inferno 作为分布式资源管理的工具, 通过 Inferno 的 shell 脚本语言作为开发语言来构建集群并处理并行框架问题. 具体的 Inferno shell 的用法我们参照 Roger Peppé 的《The Inferno Shell》⁵一文进行运用. 值得一提的是 Inferno 的 shell 非常强大, 它核心功能精悍且可以动态加载第三方模块, 这一点和 python 非常相似.

关于 Inferno 的 sytx 通信协议方面, 我们调研到 Rob Pike 的关于《The Styx Architecture for Distributed Systems》⁶的工作, 了解 inferno 通过“资源即文件”的思想, 将存储设备、进程、服务、网络通信等表示为文件, 并通过文件的读、写、执行来控制各种资源或者进行交互. 而 sytx 则是作为简单而统一的网络通信协议, 构成了在系统内通信、操作资源架构的核心.

1.4.2 常见并行框架概况

由于我们小组的主要任务还是对并行框架的构建, 所以对于各个平台、模型、系统下的分布式计算和并行框架的调研是必不可少的. 通过对文献及网络资源的查找, 我们小组发现还没有在 Inferno 系统上构造并行框架的相关研究, 所以创新性抑或是独创性是有所保证的.

接下来我们小组对几个比较常用的并行框架进行了一定的调研:

³<http://www.vitanuova.com/Inferno/docs.html>

⁴http://doc.cat-v.org/Inferno/4th_edition

⁵www.vitanuova.com/Inferno/papers/sh.html

⁶www.vitanuova.com/Inferno/papers/styx.html

1. MPI : Message Passing Interface (MPI) 是一个语言独立的对并行计算机进行编程的通信协议或通信系统.MPI 支持点对点或者集群通信, 具有标准化和可移植性的特点.”MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation”⁷. 在传输层 MPI 利用了 sockets 和 TCP.MPI 模型也因其高性能、可移植性、可扩展性而广泛应用于高性能计算中.
2. OpenMP : Open Multi-Processing 是一种支持多平台进行 shared memory multiprocessing programming 的应用程序接口.OpenMP 的核心元素包括进程创建、负载分配、数据环境管理、进程同步等. 除了其优秀的跨平台能力外, 在集群中 OpenMP 还可与 MPI 模型进行混合处理问题.

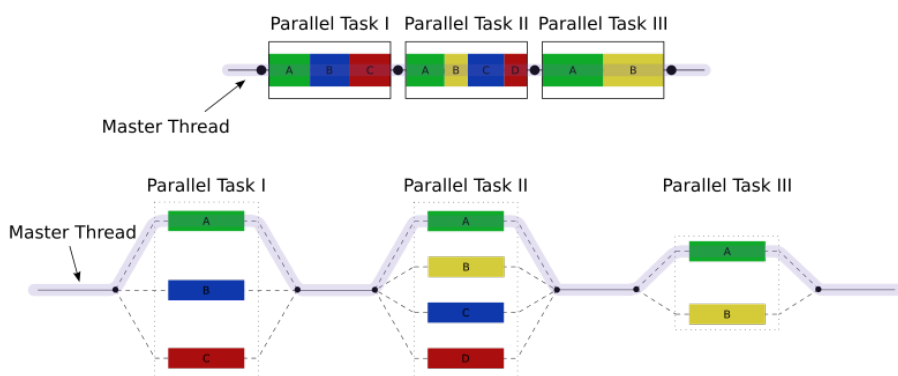


Figure 1.3: OpenMP 原理

3. CUDA : Compute Unified Device Architecture(CUDA) 是由英伟达公司创造的, 用于 GPU 上的平行处理平台. 利用 CUDA,GPU 不仅可以进行图形处理, 还可进行其他的通用计算 (所谓的 GPGPU) .
对于操作系统, 它支持 windows、linux、mac os 三大主流系统.CUDA 为开发者提供了虚拟指令集和并行计算单元的内存来进行运算.

4. TBB (Intel Threading Building Blocks): 是一个由英特尔公司开发的

⁷Gropp, William; Lusk, Ewing; Skjellum, Anthony (1996). "A High-Performance, Portable Implementation of the MPI Message Passing Interface". Parallel Computing. CiteSeerX: 10.1.1.102.9485

C++ 模板库，主要目的是使得软件开发者更好的利用多核处理器. 该库为开发者提供了一些线程安全的容器和算法，使得开发者无需过多关注系统线程的创建、同步、销毁等操作，将精力集中于业务逻辑的并行化，可以与 OpenMP 互为补充.

比较项目	OpenMP	TBB	MPI
并行粒度	线程	线程	进程
内存模式	共享内存	共享内存	分布式内存
适用环境	单机	单机	集群
通讯机制	*	*	消息传递
易用性	高	中	低
跨平台性	是	是	是
并发数据结构	不支持	支持	支持
可扩展内存分配	不支持	支持	不支持

Table 1.1: 各种并行计算框架对比

1.5 OpenCL 调研

1.5.1 简介

如何提高计算机的性能？目前在这方面主要有两种方式：第一种方式是增加处理器的核心数来支持多线程、多任务，从整体上提高计算机的性能。第二种方案是通过异构计算框架，利用 CPU，GPU 乃至 APU 等计算设备的计算能力来提高计算机性能。

如今异构系统越来越普遍，但各厂商一般只提供对自己设备编程的实现。而在异构系统上用同种风格的编程语言来实现异构编程并将不同设备作为统一的计算单元来处理的难度是很大的。

于是，OpenCL (Open Computing Language, 即开放式计算语言) 应运而生。OpenCL 定义了一套机制来实现硬件独立的软件开发环境，将不同类的硬件结合到同种执行环境中。在用户看来硬件层是透明的。利用 OpenCL 可以充分利用设备的并行特性，支持不同级别的并行，并且能有效映射到由 CPU，GPU，FPGA 和将来出现的设备所组成的同构或异构、单设备或多

设备的系统中。

OpenCL(Open Computing Language, 开放计算语言) 是一个由非盈利性技术联盟 Khronos Group 进行管理的异构编程框架。OpenCL 框架用于开发可以在各种设备上运行的应用程序。OpenCL 支持多种层次的并行, 可以高效映射到同构或异构的体系结构上, 比如单个或多个 CPU、GPU 和其他类型的设备等。OpenCL 提供了一个系统中设备端语言和主机端控制层两方面的定义。设备端语言可以高效映射到众多内存系统架构上。主机端语言的目标是以较低开销来高效管理复杂的并发程序。两者共同为开发人员提供了一种从算法设计高效过渡到实现的途径。

1.5.2 主机设备交互

OpenCL 平台模型定义主机和设备的角色, 并为设备提供了一个抽象的硬件模型。

在平台模型中, 一个主机协调在一个或多个 OpenCL 设备上的程序执行。平台可以被看作是厂商特定的 OpenCL API 实现。因此, 平台上的设备只限于厂商知晓如何进行交互的设备。例如, 如果选择 A 公司的平台, 他就无法与 B 公司的 GPU 进行通信。

平台模型还带来了一个抽象的设备架构, 编程人员在编写 OpenCL 代码时以它为目标。厂商将这个抽象架构映射到具体的硬件。考虑到可扩展性, 平台模型将一个设备定义为一系列的计算单元 (compute unit), 每个计算单元功能独立。计算单元又进一步划分为处理部件 (processing element)。

1.5.3 OpenCL 架构

OpenCL 由两部分组成: 第一部分是用来编写内核程序的语言, 第二部分是用来定义并控制平台的 API。

为了解决一个实际问题, 我们可以用“分治”策略将问题分解为四个模型:

平台模型 该模型描述内部单元之间的关系。主机 (host) 可以是个人计算机或超级计算机。OpenCL 设备 (device) 可以是 CPU、GPU、DSP 或其它处理器。每个 OpenCL 设备包含若干计算单元 (compute unit ,CU), 每个计算单元又由若干处理单元 (processing element ,PE) 组成。

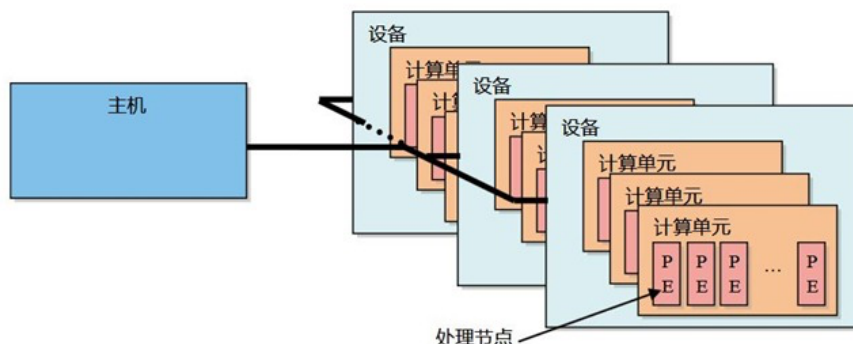


Figure 1.4: OpenCL 的平台模型 –host & device 模型

OpenCL 通过平台实现主机与设备间的交互操作。主机管理着整个平台上的所有计算资源, 所有 OpenCL 应用程序都是从主机端启动并在主机端结束的。应用程序运行时由主机提交命令 (command), 在设备上的处理单元中执行计算。每个计算单元内所有的处理单元都会执行相同的一套指令流程。每个处理单元以单指令多数据 SIMD 或单程序多数据 SPMD 模式运行指令流。

然而, 一般而言, 选定了一个平台, 那么就只能在这个平台所支持的设备上作计算了。用了 Intel SDK 就只能用 Intel 的 CPU 进行计算, 用了 APP SDK 就只能在 AMD 的 CPU 和 GPU 上进行计算。而且, 一般两个不同的平台间是不能进行通信。

执行模型 OpenCL 执行两类程序: 内核程序 (kernel) 和主机程序 (host program); 前者由若干个 OpenCL 设备执行, 后者由主机执行。OpenCL 通过主机程序定义上下文 (context) 并创建一个被称为命令队列 (command-queue) 的数据结构来管理内核程序的执行。在命令队列中, 内核程序可顺序执行 (In-order Execution) 也可乱序执行 (Out-of-order Execution)。

每当主机提交内核程序到设备上执行时, 系统便会创建一个 N 维 (N 可取 1,2,3) 的索引空间 NDRang。内核程序将索引空间中的每一点用一个工作项 (work-item) 来表示, 将若干个工作项划分成一个工作组 (work-group)。在一个计算单元内可运行同一个工作组中的工作项, 并且该组内的工作项可以并发执行在多个处理单元上。

执行模型可以进而再划分为内核，上下文，命令队列。

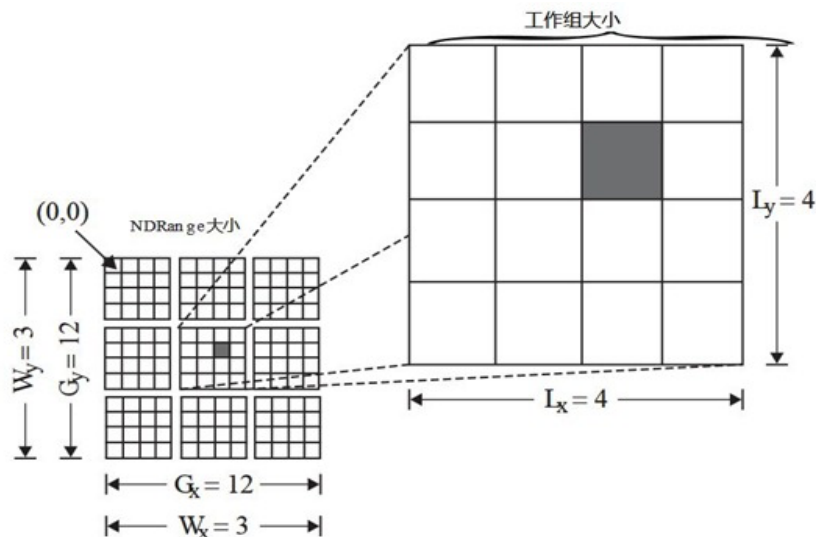


Figure 1.5: OpenCL's kernel

1. 内核: 内核是执行模型的核心，它执行在设备上。在执行内核之前，需要先指定 $NDRange$ (N 维度的范围)。 $NDRange$ 是一个一到三维的 index。这个 index 里节点的总 f 数目记为全局工作节点数。把全局工作节点分组，我们可以方便地管理各个节点。
全局工作节点数 = 工作组数 \times 工作组中节点的数目。如图 1.5，一共包含 144 个节点，分为 9 个工作组，每个工作组包含 16 个节点。
通过指定合理的节点数、工作组数，我们可以提高程序的并行度。
2. 上下文: 一个主机要使得内核运行在设备上，必须要有一个上下文来与设备进行交互。上下文是一个抽象的容器，用来管理在设备上的内存对象并且跟踪在设备上创建的程序和内核。主机通过上下文来与设备交互，使得内核运行在设备上。
3. 命令队列: 每个设备都有自己的命令队列，而主机则负责把命令发送到对应设备的命令队列上。命令队列对在设备上执行的命令进行调度。在主机和设备上，命令是异步执行的。

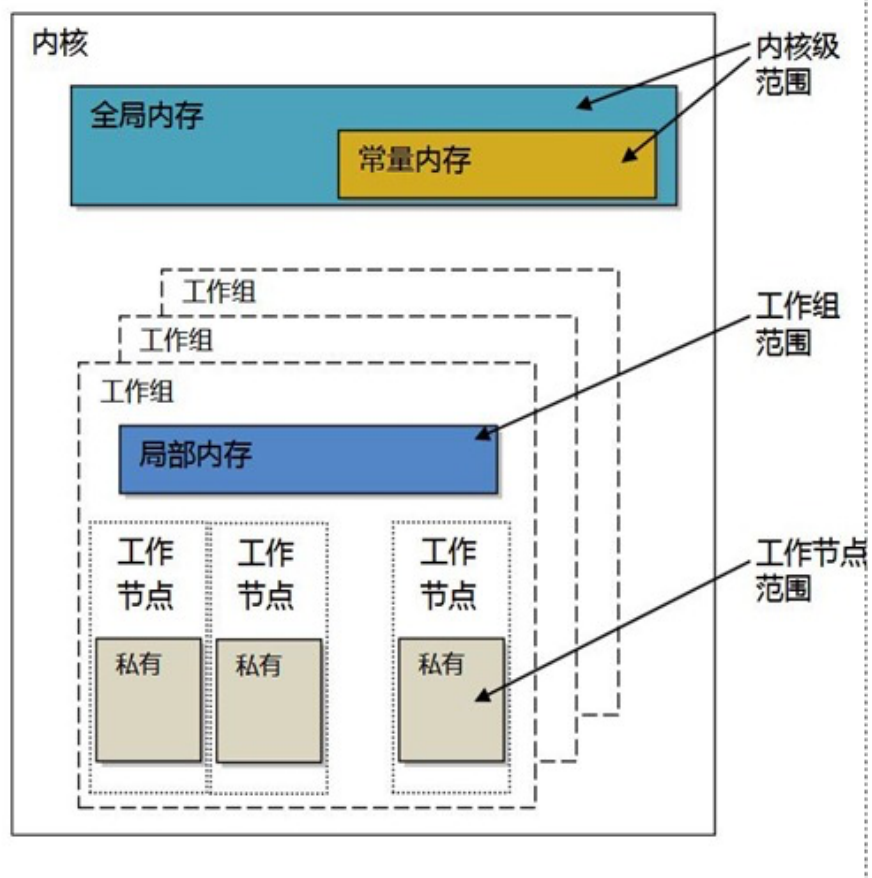


Figure 1.6: OpenCL 的内存模型

内存模型

OpenCL 把内存模型抽象化了, 通过这样来方便用户编写程序。在图 1.6 中, 内存的分类如下:

- 全局内存: 所有节点都可以对其进行读写。
- 常量内存: 全局内存中的一块区域, 在内核的执行过程中保持不变。由主机来负责分配和初始化其中的内存对象。
- 局部内存: 由一个工作组里各个工作节点共享的内存区域。在 OpenCL 设备上, 它可能会作为一块专有的内存区域存在, 也可能被映射到全局内存中。
- 私有内存: 仅属于一个工作节点的内存区域。

编程模型 数据并行和任务并行是 OpenCL 可以支持的两种并行编程模型, 同时两者的混合模型也得到支持。通常情况下, OpenCL 采用的首要模型是数据并行, 而对多核 CPU 主要采用任务并行。

数据并行编程模型中, 一系列的指令会作用到内存对象 (memory object) 的多个元素上。严格来说, 数据并行要求内存对象单元与工作项保持一对一的映射, 而在实际应用中, 并不要求严格按照这种方式。在数据并行编程模型中, OpenCL 又提供了一种分级方式, 有两种方法: 显式分级模型和隐式分级模型; 前者要求开发人员指出工作项的总数和工作项所属的工作组; 而后者仅需要开发人员定义工作项的总数, 对于工作项的划分则根据 OpenCL 的实现来管理。在任务并行编程模型上, 每个工作项都相当于在一个单的计算单元内, 该单元内只有单一工作组, 该工作组只有该工作项本身在执行。

OpenCL 软件框架 OpenCL 软件框架包含三部分: OpenCL 平台层、OpenCL 运行时和 OpenCL 编译器。在 OpenCL 平台层上, 开发人员可以查询系统中的平台数目并选定运行平台, 在指定的平台上选择必要的计算设备并对它们进行初始化, 然后可以建立上下文, 并创建命令队列。执行内核程序、读、写及复制缓冲区和同步操作等都是通过命令队列中的命令实现的。一个命令队列和一个 OpenCL 设备是一对一的关系。在 OpenCL 运行时中, 开发人员建立内核实例, 并将其映射到正确的内存空间中, 接着在命令队列中排队执行内核。OpenCL 编译器负责编译运行在设备上的程序, 并创建可执行程序。

OpenCL 实现原理 从编程的角度来说, OpenCL 中的操作都是和一个给定的上下文环境有关, 每个上下文环境中都有若干个相关的设备。在上下文环境中, OpenCL 能够保证设备之间的内存一致性。OpenCL 使用 buffers(1 维的内存块) 和 images(2 维和 3 维的内存块) 来存储内核数据。一旦分配了存放内核数据的内存空间, 并且指定了运行内核程序的设备, 就需要装载和编译内核程序。为了执行内核程序, 程序员必须建立一个内核对象 (kernel object), 并且设置内核参数。之后, 内核就可以在命令队列中排队执行了。通常情况下, OpenCL 设计的大致流程为:

- 创建并初始化 OpenCL 设备和上下文环境, 建立命令队列;
- 创建并编译源程序, 建立内核句柄;
- 分配数据所需内存空间, 并将数据复制到 OpenCL 设备上;
- 设置内核参数;
- 执行内核程序;
- 将计算结果从 OpenCL 设备复制到主机中;
- 释放系统所占资源。

1.6 MPI 调研

MPI 是一种消息传递编程模型并可以被广泛使用的编写消息传递程序的标准。它由一组库函数组成, 在 Fortran 或 C 的基础上扩展为一种并行程序设计语言。严格来说,MPI 只是一个库, 提供了应用程序的编程接口, 方便 Fortran77/90 或 C/C++ 等编程语言直接对这些库例程或函数进行调用, 实现进程间通信。

1.6.1 MPI 数据类型

为了支持异构环境和易于编程,MPI 定义了精确的数据类型参数而不使用字节计数, 以数据类型为单位指定消息的长度, 这样可以独立于具体的实现, 并且更接近于用户的观点。对于 C 和 Fortran,MPI 均预定义了一组数据类型 (如 MPI_INT、MPI_DOUBLE 等) 和一些附加的数据类型 (如 MPI_LONG_LONG_INT 等), 这些与语言中的数据类型相对应。MPI 除

除了可以发送或接收连续的数据之外,还可以处理不连续的数据,如多维向量或类 C 语言中的结构数据类型。为了发送不连续的数据,MPI 定义了打包 (pack) 与解包 (un-pack) 操作,在发送前显式地把数据包装到一个连续地缓冲区,在接收之后再从连续缓冲区中解包。MPI 还允许发送和接收不同的数据类型,使得数据重映射方便灵活。通过使用不同的数据类型调用 MPI_SEND ,可以发送不同类型的数据,使得 MPI 对数据的处理更为灵活。

1.6.2 通信域

通信域是 MPI 的一个关键概念,它以对象形式存在,作为通信操作的附加参数。通信域为开发消息传递程序提供了模块化支持,从而强有力地支持开发并行库和大规模代码。MPICH 中的一个通信域定义了一组进程和一个通信的上下文,虚拟处理器拓扑、属性等内容。通信上下文是通信域所具有的一个特性,它允许对通信空间进行划分,提供了一个相对独立的通信区域,由系统严格管理,对用户是透明的,有力地保证了库代码的消息通信互不干扰。MPI_COMM_WORLD 是一个由 MPI 提供的预定义的通信域,它包括所有的进程。所有的 MPI 实现都要求提供 MPI_COMM_WORLD 通信域,在进程的生命期中不允许将其释放。用户也可以在原有通信域的基础上,定义新的通信域。通信域为库和通信模式提供了一种重要的封装机制。

1.6.3 MPI 调用接口

MPI-1 提供了 128 个调用接口,MPI-2 提供了 287 个调用接口,但 MPI 所有的通信功能可以用它的 6 个基本调用来实现,也就是说可以用这 6 个基本调用来完成基本的消息传递并行编程。

1. MPI 初始化:MPI_INIT() 是 MPI 程序的第一个调用,它完成 MPI 程序所有的初始化工作,也是 MPI 程序的第一条可执行语句。
2. MPI 结束:MPI_FINALIZE() 是 MPI 程序的最后一个调用,它结束 MPI 程序的运行,也是所有 MPI 程序的最后一条可执行语句。
3. 获取当前进程标识:MPI_COMM_RANK(comm ,rank) 返回调用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。指定的通

信域内每个进程分配一个独立的进程标识序号, 例如有 n 个进程, 则其标识为 $0 \sim n-1$ 。

4. 通信域包含的进程数: `MPI_COMM_SIZE(comm, size)` 返回给定通信域中所包括的进程的个数, 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。
5. 消息发送: `MPI_SEND(buf, count, datatype, dest, tag, comm)`: `MPI_SEND` 将发送缓冲区 `buf` 中的 `count` 个 `datatype` 数据类型的数据发送到目的进程 `dest`, `tag` 是个整型数, 它表明本次发送的消息标志, 使用这一标志, 就可以把本次发送的消息和本进程向同一目的进程发送的其他消息区别开来。 `MPI_SEND` 操作指定的发送缓冲区是由 `count` 个类型为 `datatype` 的连续数据空间组成, 起始地址为 `buf`。
6. 消息接收: `MPI_RECV(buf, count, datatype, source, tag, comm, status)` 从指定的进程 `source` 接收消息, 并且该消息的数据类型和消息标识和本接收进程指定的 `datatype` 和 `tag` 相一致, 接收到的消息所包含的数据元素的个数最多不能超过 `count`。 `status` 是一个返回状态变量, 它保存了发送数据进程的标识、接收消息的大小数量、标志、接收操作返回的错误代码等信息。接收到的消息的源地址、标志以及数量都可以从变量 `status` 中获取, 如在 C 语言中, 通过对 `status.MPI_SOURCE`, `status.MPI_TAG`, `status.MPI_ERROR` 引用, 可以得到返回状态中所包含的发送数据进程的标识, 发送数据使用的 `tag` 标识和本接收操作返回的错误代码。

1.6.4 MPI 并行编程模式

MPI 具有两种最基本的并程序序设计模式: 对等模式和主从模式, 大部分并程序序都是这两种模式之一或二者的组合。并程序序设计的两种基本模式可概括出程序各个部分的关系。对等模式中程序的各个部分地位相同, 功能和代码基本一致, 只是处理的数据或对象不同。而主从模式体现出程序通信进程之间的一种主从或依赖关系。并行算法是并行计算的核心, 用 MPI 实现并行算法, 这两种基本模式基本上可以表达用户的要求, 对于复杂的并行算法, 在 MPI 中都可以转换成这两种基本模式的组合或嵌套。

主从模式 主从模式又称为 Master/Slave 模式, 在这种模式的并程序中, 存在一个单独执行控制程序的主进程 (Master), 负责任务的划分、分派、结果的收集, 负责所有进程间的协调和网络调度, 并且在其它结点机调用 Slave 程序等. 执行从程序的若干进程称为从进程 (Slave), 负责子任务的接收、计算和结果的发送.

在这种模式下, 一个程序由两部分构成: Master 程序和 Slave 程序. 主进程执行 Master 程序, 各子进程分别执行各自的 Slave 程序. 主进程通过消息传递通信函数完成与并行处理机节点上运行的各子进程间的数据传输. Master 程序和 Slave 程序各自的程序框架如下:

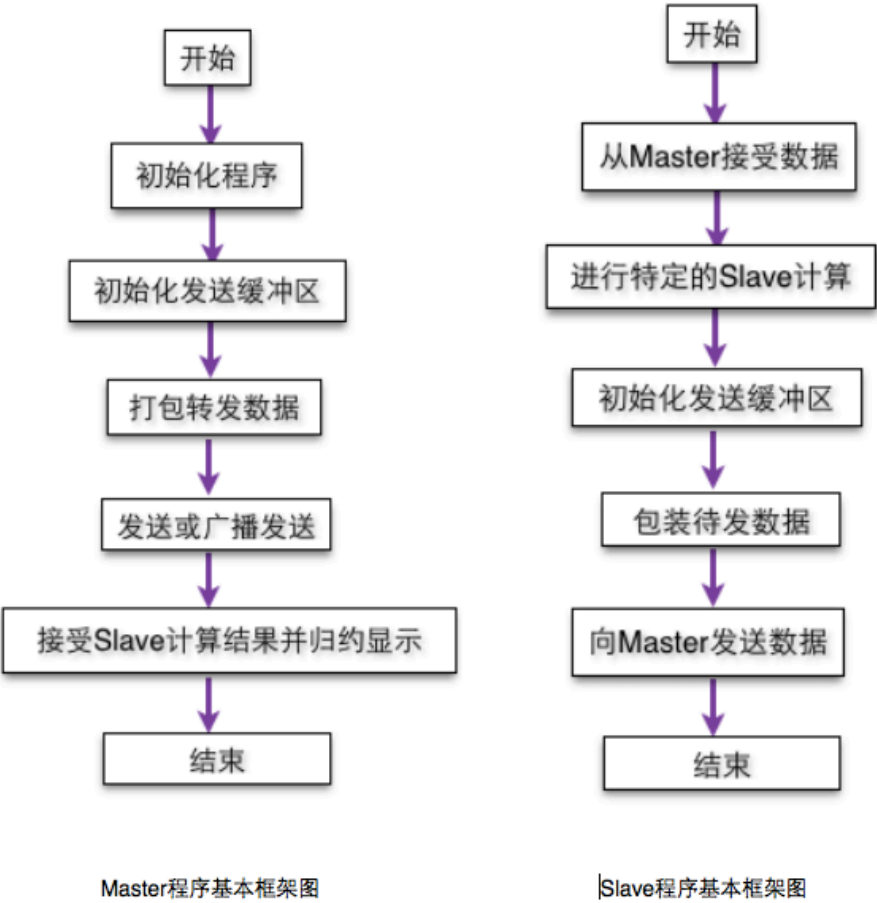


Figure 1.7: MPI 主从模式

对等模式 对等模式又称为 SPMD 模式, 在这种模式的并程序中, 各个部分地位相同, 所有进程执行同一个程序, 只是各进程计算的数据不同, 控制和处理都在计算节点上. 各节点使用消息实现同步或异步通讯. 对等模式程序基本框架如下:

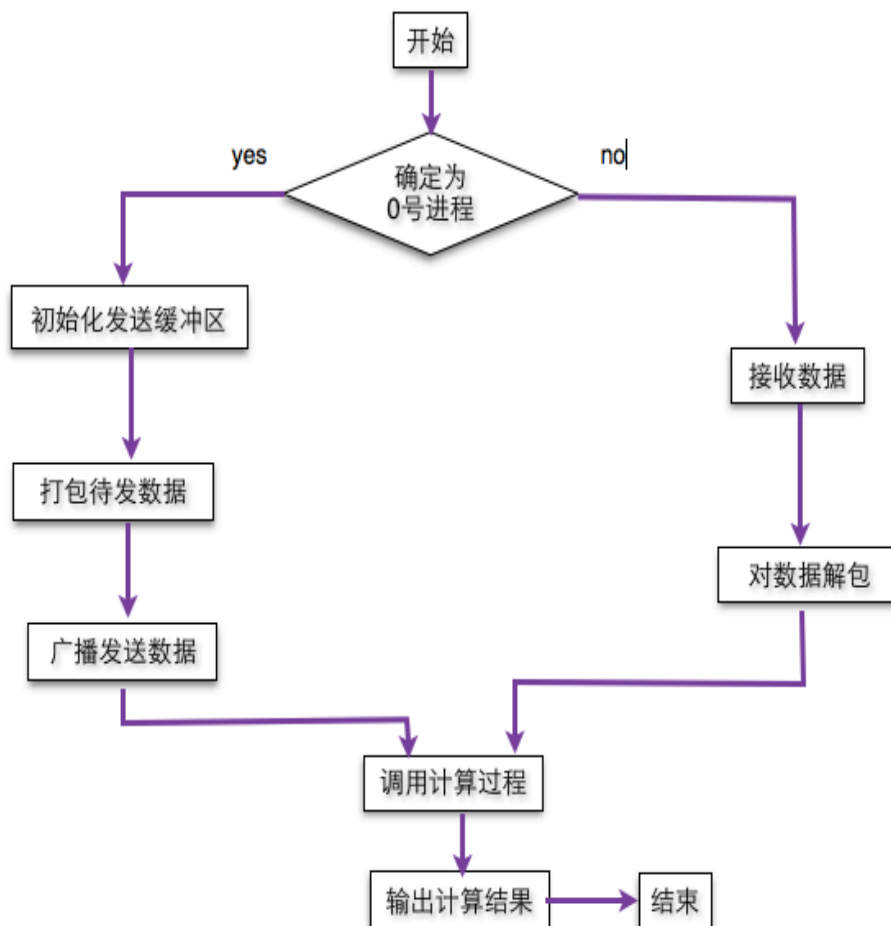


Figure 1.8: MPI 对等模式

1.7 想法来源及可能遇到的挑战

并行计算如果按照硬件环境进行分类, 那么以分布式计算为基础的计算机集群使其中典型的一类. 以上三种并行框架大多是跨平台的, 而 inferno

自身却具有分布式系统和以“资源皆文件”的优势,使得并行计算框架的构建更具有可执行性,但这个方向还鲜有相关研究.

根据相关调研,我们可能会面对下述问题:

1. 计算机集群实现分布式运算的特点,计算过程不一定是同步的,所以负载均衡是一个有挑战的问题,对应解决方案有 Round-robin DNS,scheduling algorithms 等.
2. 由于分布式计算需要共享实时数据,对于网络带宽以及低延迟网络的需求也是其需要考虑的问题,可以运用更高效率的传输媒介来处理.
3. 因为考虑的是并行框架,各个计算机、各个数据以及各个指令之间的竞争需要处理.例如:当计算机尝试覆盖相同或者旧的数据,而此时就的数据仍在被读取.结果可能是下面的一个或者多个情况:机器宕机、出现非法操作并结束程序、错误的读取旧数据、或者错误的写入新数据.在这里,竞争的危害出现于通过访问同一个信息通道的数据.解决的方法可以通过标记正在处理的数据以及标记读写来处理.
4. 分布式系统中的另一个问题是容错性问题(fault-tolerance).这也是有名的拜占庭将军问题(Byzantine failures).在容错的分布式计算中,拜占庭失效可以是分布式系统中算法执行过程中的任意一个错误.这些错误被统称为“崩溃失效”和“发送与遗漏是失效”.当拜占庭失效发生时,系统可能会做出任何不可预料反应.这些任意的失效可以粗略地分成以下几类;进行算法的另一步时失效,即崩溃失效:无法正确执行算法的一个步骤;执行了任意一个非算法指定的步骤.具体来说就是所写的并行框架在没有容错处理时出现让人匪夷所思的随机错误.具体可以通过口头消息算法或签名消息算法解决.

2 可行性研究报告

2.1 理论依据与技术依据

2.1.1 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端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端间的消息（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.

2.1.2 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。

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 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.

2.2 创新点

2.2.1 良好的异构支持

由于 Inferno 的分布式特性. 我们的并行框架具有天然的跨平台性, 可以实现 arm, x86, amd64 节点混合成的混合集群.

同时我们借鉴了 OpenCL 的平台模型, 并采用抢占式任务分发以及细并行粒度的任务分配, 可以使得计算能力相差很大的节点协作计算. 具体内容可以参见概要设计报告中异构的部分.

2.2.2 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.

2.2.3 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.

2.2.4 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, PowerPC, 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.

2.2.5 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`. 编译器编译时则会自动会链接 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.

2.2.6 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。

2.2.7 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。

3 概要设计报告

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 `host & device` 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。
- 在 Inferno 在 x86/64 的机子上完成部署。并将 Inferno 移植到 Raspberry Pi 上，搭建通过局域网互联的跨架构的集群。
- 在搭建的集群上测试我们解释器，完成最终部署。

3.2 平台模型 - OpenCL 的 host & device 模式

根据我们之前对常见并行计算框架的调研以及 Inferno 的架构，我们最终决定选用 Opencl 所用的平台模型 - host & device 模型作为我们的平台模型。原因如下：

- host & device 模式非常成熟。当前最为流行的支持异构的并行计算框架 OpenCL 和 CUDA 都采用它作为自己的平台模型
- 在 host & device 模式中，host 就像一个大管家，完成任务调度，节点通讯，负载均衡功能。因此 device 则只需 care 自己那部分的运算即可，无需 care 自身目前所处的环境。即实际运行环境对 device 是不可见的，device 不需要对运行环境的变化做出变化。这一点对于异构是至为重要的。
- Inferno 的架构非常适合采用 host & device 模式。根据前期对 Inferno 的 namespace 和 styx 的调研，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。所以可以选择一台节点作为 host，其余节点作为 device。然后将 device 全部挂载在 host 上。由 host 分发任务，响应 device 请求，进行 device 间调度等。采用这种平台模型可以大大推进我们开发速度。

3.3 异构性

首先，我们采用了 OpenCL 和 CUDA 所使用的平台模型 - host & device 模型作为我们的平台模型，奠定了异构基础。

为了进一步支持，我们提出了细并行粒度和抢占式任务分配的观点。

3.3.1 抢占式任务分配

为了更好地支持异构，我们决定采用抢占式任务分配的方式。在这种方式下，host 无需主动分配任务给 device，而是将下一个任务放在共享池中让已经完成了之前任务的空闲 device 来抢占。当前共享池中的任务被抢占后，host 根据计算情况再将新的任务放在共享池中。

3.3.2 细并行粒度

在并行计算中，并行粒度 = 计算量 / 通信量。计算和通信交替出现，两者通过同步时间相区分。并行粒度受限于应用程序算法的内在特性。由于选择正确的并行粒度有助于程序暴露更多的可并行性，因此为了充分发挥基础平台的性能，选择正确的并行粒度非常重要。选择合适的并行粒度可以提高并行化加速比。经过调研我们得到如下结论：

细粒度的并行

- 计算强度低
- 没有足够的任务来隐藏长时间的异步通信
- 容易通过提供大量可管理的 (即更小的) 工作单元来实现负载均衡

粗粒度的并行

- 计算强度高
- 完整的应用可以作为并行的粒度
- 难以有效实现负载均衡

根据上述结论，我们决定采用细并行粒度。对于每个 device，由于并行粒度细，各个 device 之间同步需求不大，且每次由 host 分配给它的任务并不繁重，可以在较短时间计算完成。再由抢占式任务分配机制，计算完成的节点把结果提交给 host 之后可以直接从 host 中抢占任务，而不是等待其余 device。

如此实现，在一定程度上可以避免计算能力低的节点拖累计算能力高的节点，从而提高了异构性。

3.4 Raspberry Pi2

Inferno 强调的是分布式，适用于多种架构，从 pc 到嵌入式系统。所以在设计并行框架时应该利用 Inferno 这一强大的特性实现对异构的支持。于是搭建 Inferno 集群时，我们准备安排一个 arm 架构的结点。



Figure 3.1: Raspberry Pi2

经过调研论证，我们决定选择 Raspberry Pi2 作为 arm 节点。

Raspberry Pi 是一款基于 Linux 系统的只有一张信用卡大小的单板机电脑。它由英国的树莓派基金会所开发，目的是以低价硬件及自由软件刺激在学校的基本的电脑科学教育。

2015 年 2 月，树莓派基金会发布了第二代产品 - Raspberry Pi2，售价 35 美元。Raspberry Pi2 采用 4 核 Broadcom BCM2836 (ARMv7-A) 芯片、双核 VideoCore IV GPU 和 1GB 内存，

4 理论研究与实践

纸上得来终觉浅，绝知此事要躬行。我们组经过反复论证探讨之后，开始实践工作。

4.1 Inferno 在 Raspberry Pi 移植

根据前期调研，我们决定采用 hosted 方式在 Raspberry Pi 上运行 inferno。刚好组长阮震元所在的 ACSA 实验室里有空闲的树莓派可以使用。

首先我们在自己的电脑上搭建了 arm 的交叉编译工具链条。根据 Raspberry Pi 的情况我们采用了 armv6j-hardfloat-linux-gnueabi。完成了 arm-glibc, arm-gcc 等交叉编译环境的部署。

接着即为针对 Raspberry Pi 的环境对 inferno 源码做了 11 处修改, 使得原版的 Inferno 与 Raspberry Pi 适配。

进行的过程中遇到了各种困难, 例如 Inferno 原版的 setfcr-Linux-arm.c 已经过时了, 在使用 4.7 以上的 arm-gcc 会遇到错误, 需要对其根据编译环境做相应调整, 又如 Inferno 的 styx 协议无法在 Raspberry Pi 上正常运行, 最终定位到内核 AF_INET 模块, 需要对内核配置进行修改。

在移植的过程中, 我们还积极联系了 Inferno 的官方开发组, 和他们共同完成这项工作。在此阶段组长阮震元还向 Inferno 官方上报了 2 个 Inferno 在 arm 下的 bug, 均引起重视。一个被官方列为 major 等级⁸, 另一个则被官方列为 critical 等级⁹。阮震元还针对 Inferno 在 amd64 下编译问题向官方提供了一个有效的 patch¹⁰。

最终我们在 Raspberry Pi 上成功运行了 Inferno!

4.2 初步尝试 — 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
    }
}
```

⁸<https://bitbucket.org/inferno-os/inferno-os/issue/326/arm-listen-failed-because-ipv6-support-is>

⁹<https://bitbucket.org/inferno-os/inferno-os/issue/325/arm-build-failed>

¹⁰<https://bitbucket.org/inferno-os/inferno-os/issue/329/a-patch-for-x64>

```

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.

4.3 初步尝试 – 改写的 `host & device` 式并行代码

```
host 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

```

device 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

```

4.4 file2chan - sharedPool 的基础

file2chan 是 user namespace 下的程序，通过对 Inferno 内核的系统调用实现了内存中的虚拟文件。

file2chan 接受 2 个代码块 writcmd 和 readcmd 作为参数。当用户读虚拟文件时则会执行 readcmd 中的代码，当用户对虚拟文件写时则会执行 writcmd 中的代码。

利用 file2chan 我们将虚拟出一个虚拟文件并进一步封装成一个 shared-Pool 用来做动态的节点间进程通讯。

file2chan 的代码如下

```

implement Shellbuiltin;

include "sys.m";
    sys: Sys;
include "draw.m";
include "lock.m";
    lock: Lock;
    Semaphore: import lock;
include "sh.m";
    sh: Sh;
    Listnode, Context: import sh;

```

```

    myself: Shellbuiltin;

Tag: adt {
    tagid, blocked: int;
    offset, fid: int;
    pick {
    Read =>
        count: int;
        rc: chan of (array of byte, string);
    Write =>
        data: array of byte;
        wc: chan of (int, string);
    }
};

taglock: ref Lock->Semaphore;
maxtagid := 1;
tags := array[16] of list of ref Tag;

initbuiltin(ctxt: ref Context, shmod: Sh):
    string
{
    sys = load Sys Sys->PATH;
    sh = shmod;

    myself = load Shellbuiltin "$self";
    if (myself == nil)
        ctxt.fail("bad module", sys->sprint("
            file2chan: cannot load self: %r"));

    lock = load Lock Lock->PATH;
    if (lock == nil) ctxt.fail("bad module",
        sys->sprint("file2chan: cannot load %s:

```

```

        %r ", Lock->PATH));
lock->init();

taglock = Semaphore.new();
if (taglock == nil)
    ctxt.fail("no lock", "file2chan: cannot
        make lock");

ctxt.addbuiltin("file2chan", myself);
ctxt.addbuiltin("rblock", myself);
ctxt.addbuiltin("rread", myself);
ctxt.addbuiltin("rreadone", myself);
ctxt.addbuiltin("rwrite", myself);
ctxt.addbuiltin("rerror", myself);
ctxt.addbuiltin("fetchwdata", myself);
ctxt.addbuiltin("putrdata", myself);
ctxt.addsbuiltin("rget", myself);

return nil;
}

whatis(nil: ref Sh->Context, nil: Sh, nil:
    string, nil: int): string
{
    return nil;
}

getself(): Shellbuiltin
{
    return myself;
}

```

```

runbuiltin(ctxt: ref Context, nil: Sh,
           cmd: list of ref Listnode, nil: int
           ): string
{
  case (hd cmd).word {
    "file2chan" =>      return
      builtin_file2chan(ctxt, cmd);
    "rblock" =>        return builtin_rblock(ctxt,
      cmd);
    "rread" =>          return builtin_rread(
      ctxt, cmd, 0);
    "rreadone" =>       return builtin_rread(
      ctxt, cmd, 1);
    "rwrite" =>         return builtin_rwrite(ctxt,
      cmd);
    "rerror" =>         return builtin_rerror(ctxt,
      cmd);
    "fetchwdata" =>     return builtin_fetchwdata(
      ctxt, cmd);
    "putrdata" =>       return builtin_putrdata
      (ctxt, cmd);
  }
  return nil;
}

runsbuiltin(ctxt: ref Context, nil: Sh,
            argv: list of ref Listnode): list
            of ref Listnode
{
  # could add ${rtags} to retrieve list of
    currently outstanding tags
  case (hd argv).word {
    "rget" =>           return sbuiltin_rget(

```

```

        ctxt, argv);
    }
    return nil;
}

builtin_file2chan(ctxt: ref Context, argv: list
of ref Listnode): string
{
    rcmd, wcmd, ccmd: ref Listnode;
    path: string;

    n := len argv;
    if (n < 4 || n > 5)
        ctxt.fail("usage", "usage: file2chan
            file {readcmd} {writecmd} [ {
                closecmd} ]");

    (path, argv) = ((hd tl argv).word, tl tl
        argv);
    (rcmd, argv) = (hd argv, tl argv);
    (wcmd, argv) = (hd argv, tl argv);
    if (argv != nil)
        ccmd = hd argv;
    if (path == nil || !iscmd(rcmd) || !iscmd(
        wcmd) || (ccmd != nil && !iscmd(ccmd)))
        ctxt.fail("usage", "usage: file2chan
            file {readcmd} {writecmd} [ {
                closecmd} ]");

    (dir, f) := pathsplit(path);
    if (sys->bind("#s", dir, Sys->MBEFORE|Sys->
        MCREATE) == -1) {
        reporterror(ctxt, sys->sprint("

```



```

        file2chan: cannot bind #s: %r"));
    return "no #s";
}
fio := sys->file2chan(dir, f);
if (fio == nil) {
    reporterror(ctxt, sys->sprint("
        file2chan: cannot make %s: %r", path
    ));
    return "cannot make chan";
}
sync := chan of int;
spawn srv(sync, ctxt, fio, rcmd, wcmd, ccmd
);
apid := <-sync;
ctxt.set("apid", ref Listnode(nil, string
    apid) :: nil);
if (ctxt.options() & ctxt.INTERACTIVE)
    sys->fprintf(sys->fildes(2), "%d\n",
        apid);
return nil;
}

srv(sync: chan of int, ctxt: ref Context,
    fio: ref Sys->FileIO, rcmd, wcmd, ccmd:
    ref Listnode)
{
    ctxt = ctxt.copy(1);
    sync <== sys->pctl(0, nil);
    for (;;) {
        fid, offset, count: int;
        rc: Sys->Rread;
        wc: Sys->Rwrite;
        d: array of byte;

```

```

t: ref Tag = nil;
cmd: ref Listnode = nil;
alt {
(offset , count , fid , rc) = <-fio.read
=>
    if (rc != nil) {
        t = ref Tag.Read(0 , 0 , offset ,
            fid , count , rc);
        cmd = rcmd;
    } else
        continue;      # we get a
                        close on both read and write
                        ...
(offset , d , fid , wc) = <-fio.write =>
    if (wc != nil) {
        t = ref Tag.Write(0 , 0 , offset ,
            fid , d , wc);
        cmd = wcmd;
    }
}
if (t != nil) {
    addtag(t);
    ctxt.setlocal("tag", ref Listnode(
        nil , string t.tagid) :: nil);
    ctxt.run(cmd :: nil , 0);
    taglock.obtain();
    # make a default reply if it hasn't
      been deliberately blocked.
    del := 0;
    if (t.tagid >= 0 && !t.blocked) {
        pick mt := t {
            Read =>
                rreply(mt.rc , nil , "invalid

```

```

        read");
    Write =>
        wreply(mt.wc, len mt.data,
            nil);
    }
    del = 1;
}
taglock.release();
if (del)
    deltag(t.tagid);
    ctxt.setlocal("tag", nil);
} else if (ccmd != nil) {
    t = ref Tag.Read(0, 0, -1, fid, -1,
        nil);
    addtag(t);
    ctxt.setlocal("tag", ref Listnode(
        nil, string t.tagid) :: nil);
    ctxt.run(ccmd :: nil, 0);
    deltag(t.tagid);
    ctxt.setlocal("tag", nil);
}
}

builtin_rread(ctxt: ref Context, argv: list of
    ref Listnode, one: int): string
{
    n := len argv;
    if (n < 2 || n > 3)
        ctxt.fail("usage", "usage: "+(hd argv).
            word+" [tag] data");
    argv = tl argv;

```

```

    t := envgettag(ctxt, argv, n == 3);
    if (t == nil)
        ctxt.fail("bad tag", "rread: cannot
            find tag");
    if (n == 3)
        argv = tl argv;
    mt := etr(ctxt, "rread", t);
    arg := word(hd argv);
    d := array of byte arg;
    if (one) {
        if (mt.offset >= len d)
            d = nil;
        else
            d = d[mt.offset:];
    }
    if (len d > mt.count)
        d = d[0:mt.count];
    rreply(mt.rc, d, nil);
    deltag(t.tagid);
    return nil;
}

builtin_rwrite(ctxt: ref Context, argv: list of
    ref Listnode): string
{
    n := len argv;
    if (n > 3)
        ctxt.fail("usage", "usage: rwrite [tag
            [count]]");
    t := envgettag(ctxt, tl argv, n > 1);
    if (t == nil)
        ctxt.fail("bad tag", "rwrite: cannot
            find tag");

```

```

mt := etw(ctxt, "rwrite", t);
count := len mt.data;
if (n == 3) {
    arg := word(hd tl argv);
    if (!isnum(arg))
        ctxt.fail("usage", "usage: freply [
            tag [count]]");
    count = int arg;
}
wreply(mt.wc, count, nil);
deltag(t.tagid);
return nil;
}

builtin_rblock(ctxt: ref Context, argv: list of
    ref Listnode): string
{
    argv = tl argv;
    if (len argv > 1)
        ctxt.fail("usage", "usage: rblock [tag
            ]");
    t := envgettag(ctxt, argv, argv != nil);
    if (t == nil)
        ctxt.fail("bad tag", "rblock: cannot
            find tag");
    t.blocked = 1;
    return nil;
}

sbuiltin_rget(ctxt: ref Context, argv: list of
    ref Listnode): list of ref Listnode
{

```

```

n := len argv;
if (n < 2 || n > 3)
    ctxt.fail("usage", "usage: rget (data|
        count|offset|fid) [tag]");
argv = tl argv;
t := envgettag(ctxt, tl argv, tl argv !=
    nil);
if (t == nil)
    ctxt.fail("bad tag", "rget: cannot find
        tag");
s := "";
case (hd argv).word {
"data" =>
    s = string etw(ctxt, "rget", t).data;
"count" =>
    s = string etr(ctxt, "rget", t).count;
"offset" =>
    s = string t.offset;
"fid" =>
    s = string t.fid;
* =>
    ctxt.fail("usage", "usage: rget (data|
        count|offset|fid) [tag]");
}

return ref Listnode(nil, s) :: nil;
}

builtin_fetchwdata(ctxt: ref Context, argv:
    list of ref Listnode): string
{
    argv = tl argv;
    if (len argv > 1)

```

```

        ctxt.fail("usage", "usage: fetchwdata [
            tag]");
    t := envgettag(ctxt, argv, argv != nil);
    if (t == nil)
        ctxt.fail("bad tag", "fetchwdata:
            cannot find tag");
    d := etw(ctxt, "fetchwdata", t).data;
    sys->write(sys->fildes(1), d, len d);
    return nil;
}

builtin_putrdata(ctxt: ref Context, argv: list
    of ref Listnode): string
{
    argv = tl argv;
    if (len argv > 1)
        ctxt.fail("usage", "usage: putrdata [
            tag]");
    t := envgettag(ctxt, argv, argv != nil);
    if (t == nil)
        ctxt.fail("bad tag", "putrdata: cannot
            find tag");
    mt := etr(ctxt, "putrdata", t);
    buf := array[mt.count] of byte;
    n := 0;
    fd := sys->fildes(0);
    while (n < mt.count) {
        nr := sys->read(fd, buf[n:mt.count], mt
            .count - n);
        if (nr <= 0)
            break;
        n += nr;
    }
}

```

```

        rreply(mt.rc, buf[0:n], nil);
        deltag(t.tagid);
        return nil;
    }

builtin_error(ctxt: ref Context, argv: list of
    ref Listnode): string
{
    # usage: ferror [tag] error
    n := len argv;
    if (n < 2 || n > 3)
        ctxt.fail("usage", "usage: ferror [tag]
            error");
    t := envgettag(ctxt, tl argv, n == 3);
    if (t == nil)
        ctxt.fail("bad tag", "error: cannot
            find tag");
    if (n == 3)
        argv = tl argv;
    err := word(hd tl argv);
    pick mt := t {
    Read =>
        rreply(mt.rc, nil, err);
    Write =>
        wreply(mt.wc, 0, err);
    }
    deltag(t.tagid);
    return nil;
}

envgettag(ctxt: ref Context, args: list of ref
    Listnode, useargs: int): ref Tag

```



```

{
    tagid: int;
    if (useargs)
        tagid = int (hd args).word;
    else {
        args = ctxt.get("tag");
        if (args == nil || tl args != nil)
            return nil;
        tagid = int (hd args).word;
    }
    return gettag(tagid);
}

etw(ctxt: ref Context, cmd: string, t: ref Tag)
: ref Tag.Write
{
    pick mt := t {
        Write => return mt;
    }
    ctxt.fail("bad tag", cmd + ": inappropriate
        tag id");
    return nil;
}

etr(ctxt: ref Context, cmd: string, t: ref Tag)
: ref Tag.Read
{
    pick mt := t {
        Read => return mt;
    }
    ctxt.fail("bad tag", cmd + ": inappropriate
        tag id");
    return nil;
}

```

```

}

wreply(wc: chan of (int, string), count: int,
       err: string)
{
    alt {
        wc <-= (count, err) => ;
        * => ;
    }
}

rreply(rc: chan of (array of byte, string), d:
       array of byte, err: string)
{
    alt {
        rc <-= (d, err) => ;
        * => ;
    }
}

word(n: ref Listnode): string
{
    if (n.word != nil)
        return n.word;
    if (n.cmd != nil)
        n.word = sh->cmd2string(n.cmd);
    return n.word;
}

isnum(s: string): int
{
    for (i := 0; i < len s; i++)
        if (s[i] > '9' || s[i] < '0')

```

```

        return 0;
    return 1;
}

iscmd(n: ref Listnode): int
{
    return n.cmd != nil || (n.word != nil && n.
        word[0] == '}');
}

addtag(t: ref Tag)
{
    taglock.obtain();
    t.tagid = maxtagid++;
    slot := t.tagid % len tags;
    tags[slot] = t :: tags[slot];
    taglock.release();
}

deltag(tagid: int)
{
    taglock.obtain();
    slot := tagid % len tags;
    nwl: list of ref Tag;
    for (wl := tags[slot]; wl != nil; wl = tl
        wl)
        if ((hd wl).tagid != tagid)
            nwl = hd wl :: nwl;
        else
            (hd wl).tagid = -1;
    tags[slot] = nwl;
    taglock.release();
}

```

```

gettag(tagid: int): ref Tag
{
    slot := tagid % len tags;
    for (wl := tags[slot]; wl != nil; wl = tl
        wl)
        if ((hd wl).tagid == tagid)
            return hd wl;
    return nil;
}

pathsplit(p: string): (string, string)
{
    for (i := len p - 1; i >= 0; i--)
        if (p[i] != '/')
            break;
    if (i < 0)
        return (p, nil);
    p = p[0:i+1];
    for (i = len p - 1; i >=0; i--)
        if (p[i] == '/')
            break;
    if (i < 0)
        return (".", p);
    return (p[0:i+1], p[i+1:]);
}

reporterror(ctxt: ref Context, err: string)
{
    if (ctxt.options() & ctxt.VERBOSE)
        sys->fprint(sys->fildes(2), "%s\n", err
            );
}

```

4.5 sharedPool - 节点间通讯 (任务分配收发池)

首先在 host 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 device 想要向 host 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 device 计算完成, 要把结果反馈给 host 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 device 向 host 提交的结果计入总结果.

4.6 device 部分代码

并行部分的 device 代码和之前的 deviceSerial 的主要区别就是任务的分配方式不同. 串行版的 deviceSerial 是自己决定计算任务, 而并行版的 device 是向 host 抢占任务, 待抢占到的任务计算完毕后则将结果放回 sharedPool, 而 device 则无需关心结果该如何统计这一切都由 host 完成, device 只需再向 host 抢占任务即可.

由于已经对 sharedPool 做了高度封装, device 只需简单调用即可. 抢占任务由 subfn getTask 实现, 提交结果由 subfn writeRequest 实现.

4.7 host 部分代码

host 部分主要实现了对 sharedPool 的封装. fn checkHost 和 fn ready-Host 负责启动各个 device 节点, 使其进入 pending 状态. file2chan /tmp/sharedPool 实现了对 sharedPool 的封装, 内部的 lock 段代码再结合外部的 subfn cannotWrite 实现了对 sharedPool 的同步处理.

4.8 并行框架开放三个函数

4.8.1 map

map 函数即所谓的映射函数，其原型来自于函数式编程。而为了说明方便，我们用 Python 和 Haskell 中的 map 函数作为比对。对于 Python，map 函数需要两个参数，一个是某个函数 f ，另一个是一个可迭代的参数 $iterable$ （通常是一个数组）。当调用 $map(f, iterable)$ 时， f 会作用于 $iterable$ 中的每一个元素并返回一个和 $iterable$ 大小相同的数组 $result$ ， $result$ 数组的每一个元素都是 f 作用于 $iterable$ 对应元素的关于 f 函数的像，即 $result[i] = f(iterable[i])$ 。举个例子：

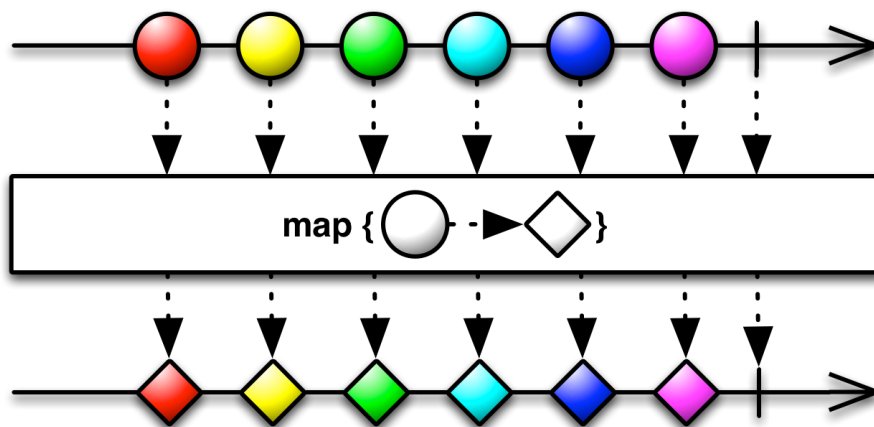


Figure 4.1: map 函数示意图

```
>>> def add100(x):  
...   return x+100  
...  
>>> iterable = [11,22,33]  
>>> map(add100, iterable)  
result = [111, 122, 133]
```

不难看出，在一般情况下，在一般含义下，map 函数可做如下解释：

$map(f, iterable) \longrightarrow [f(x) \text{ for } x \text{ in } iterable]$

对于 Haskell，也是类似的：

map 取一个函数和 List 做参数，遍历该 List 的每个元素来调用该函数

产生一个新的 List. 看下它的类型声明和实现:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

我们小组采用了类似于 Python 中 map 函数的用法, 并对其进行了并行化。作为用户视角, 用户只需标示出并行符号 @parallel_begin, 并且给出函数 f 和数组 array, 调用 map 函数, 并且以 @parallel_end 作为结束就可以得到并行化的代码。举个例子:

```
subfn calculate {
    result = ${expr $1 1 +}
}
array1 = 120 121 122 123 124 125 126 127 128
        129 130
@parallel_begin
map array1 calculate
@parallel_end
```

4.8.2 reduce

reduce 函数是一种对满足结合律的函数 f 进行计算的函数。Python 中的 reduce 内建函数是一个二元操作函数, 他用来将一个数据集合 (链表, 元组等) 中的所有数据进行下列操作: 用传给 reduce 中的函数 func() (必须是一个二元操作函数) 先对集合中的第 1, 2 个数据进行操作, 得到的结果再与第三个数据用 func() 函数运算, 然后重复进行二元操作, 直到最后得到一个结果。简单来说, 可以用这样一个形象化的式子来说明: $\text{reduce}(\text{func}, [1, 2, 3]) = \text{func}(\text{func}(1, 2), 3)$ 。更为形象地看, 我们可以用下图来分析。我们小组的 reduce 函数用法是这样的:

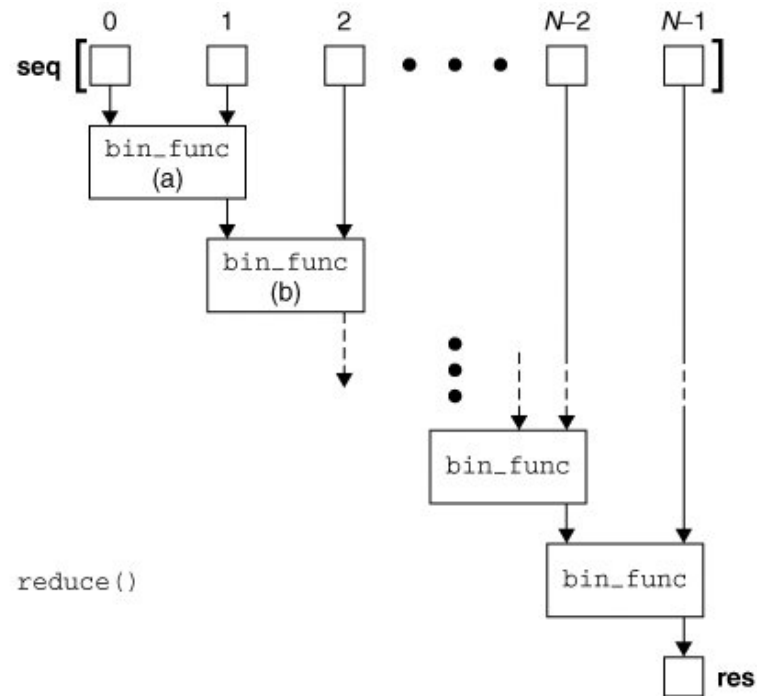
```
subfn funct {
    result = ${expr $1 $2 +}
}
```

```

input_list = 1 2 3 4 5 6 7 8 9 10 11 12 13 14
            15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
            30 31 32
@parallel_begin
reduce input_list funct
@parallel_end

```

作为用户，只需要给出函数 f ， $input_list$ ，标示出并行符号 `@parallel_begin` 并且以 `@parallel_end` 作为结束就可以得到并行化的代码，得到最后的结果。



- (a) The value of this result is `bin_func(seq[0], seq[1])`
 (b) The value of this result is `bin_func(bin_func(seq[0], seq[1]), seq[2]), etc.`

Figure 4.2: reduce 函数的示意图

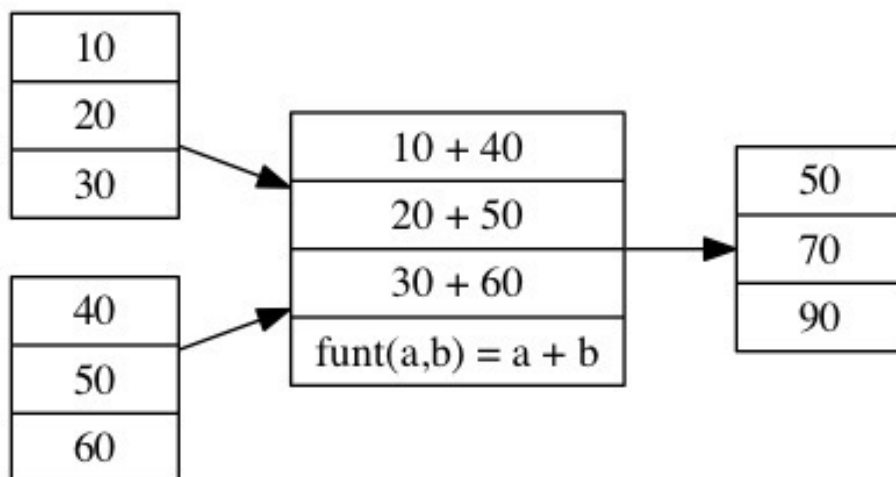


Figure 4.3: merge 函数的示意图

4.8.3 merge

merge 函数是对多个数组或者表结构进行合并的函数。我们给出一个二元函数 func, 并给出两个数组 list1,list2, 当调用 merge 的时候, 会返回另一个表 list3。简单地讲, 就是 $\text{list3}[i] = \text{func}(\text{list1}[i], \text{list2}[i])$ 。更为形象地说, 可由下图表示:

我们小组的 merge 函数用法是这样的:

```
subfn funct {
    result = ${expr $1 $2 +}
}

input_list1 = ${expr 1 99 seq}
input_list2 = ${expr 99 1 seq}

@parallel_begin
merge input_list1 input_list2 funct output_list
@parallel_end
```

作为用户, 只需要给出函数 f, input_list1, input_list2, 标示出并行符号 @parallel_begin 并且以 @parallel_end 作为结束就可以得到并行化的代

码，得到最后的合并的表。

4.9 函数并行化原理与容错机制

为了更好地体现分布式系统可靠性的特点，我们所做的并行计算框架考虑了容错问题，即当某些计算节点因为停机或网络中断而无法继续计算，并且无法与 host 交换数据的时候，要保证计算结果的正确性。为了实现故障的发现和處理，我们采取存储冗余数据的方法进行容错。

4.9.1 map

对于 map 来说，冗余数据就是 device 的信息和 device 所抢占的数据，当 device 从 host 处获得要计算的数据时，host 会在 pool 中留下冗余数据文件，文件名包含 device 所运行的计算进程的进程号 client_pid，用来区分不同的计算节点，文件的内容存储了获得的数据和其他信息，比如 ip 地址等，用来检测并处理故障。

当 host 将任务分发完成后，pool 中至少有一个刚刚发出的任务，可能还会有许多计算还没有完成，pool 中还留有很多上述的冗余数据文件，此时仍然会有一些 device 上交计算结果，并试图获取新的计算任务，此时 host 并不终止该 device 的计算，而是从 pool 的冗余文件中挑选一个其它 device 还没有算完的任务分配给它，这样可以充分利用计算资源，避免忙等，从而减少故障修复所花费的时间。

以这种方式分配出去的任务仍然要建立冗余数据文件，用来防止该 device 出现故障。此时剩余没有算完的数据可能有很多 device 在同时计算，他们之中也许有出现故障的节点，也许只是因为计算所需时间较长，计算还没有结束。但终究有一个时刻，最先完成计算的节点会向 host 提交数据，此时，host 在清除该节点的冗余数据的同时，要检查 pool 中所有的冗余文件，凡是找到和该数据相同的文件，即表示获得了同样得到该任务的其它节点的信息，此时可以逐个在其中判断故障，判断这些节点与 host 网络的连通性，如果发现某个 device 无法和 host 通信，表示该节点出现故障，要把该节点的 IP 地址从 device_ip 文件中删除，使其不影响后续的计算。

与这次提交任务相同的冗余文件全部检查完毕后，表示这个任务成功完成，可将这些冗余文件全部删除，当这些 device 再回来提交任务的时候 host 发现它的数据冗余文件已经被删除，就知道该 device 的任务已被其它 device 完成，于是放弃掉它提交的结果，并分给它新的任务，如果已没有任

何新的任务，就向它发送计算结束的信息。当任务分发完毕，并且 pool 中没有任何任务数据的冗余文件时，就表示这次 map 计算的并行部分真正完成。

使用上面的方法即可实现故障的检查和处理，保证每一个任务一定有 device 来计算，不会出现某节点出现故障导致最终结果出现错误的情况，增强了计算框架的可靠性。

4.9.2 reduce

首先，因为在其它计算中如果发现故障节点就会将它的 ip 地址从 device_ip 文件中删除，这样就会避免在 reduce 的分表阶段把任务分配给已经出现故障的节点，减轻后续故障处理的负担，提高了计算资源的利用率。我们原本的预想是当发现某个 device 出现故障的时候通过把剩余任务平均分配给其它计算节点的方式，更充分地利用计算资源，但是由于 reduce 并行条件的限制和故障出现情况的复杂性，需要考虑的情况太多，任务量比较庞大。于是我们退而求其次，假设故障发生的概率不高，采用 host 处理全部故障的方式，将最初的预想作为将来的可改进点。

具体方式如下：分表阶段结束后，就会在 pool 中留下包含 device 信息和该 device 得到的表信息的数据冗余文件，用于将来的故障处理。分发任务阶段开始后，每一个小任务的分发都会留下数据冗余文件，保留中间结果，以便修复故障的时候可以从中间结果开始计算，省去重新开始计算的时间。当有 device 算完最终结果之后（因为 host 也是计算节点 device 之一，我们假设 host 不会出故障，则一定有提交计算结果的 device），host 会删除它在 pool 中的冗余文件，然后开始检查和剩余每一个 device 的网络连通性，如果发现故障，先将出现故障的 device 从 device_ip 文件中删除，然后利用中间计算结果的数据冗余文件接替故障节点计算，host 计算完成后删除故障节点所对应的数据冗余文件。因为 host 拥有 reduce 计算过程中的所有信息，所以可以方便地将计算结果整合到后面的结果处理过程中，不需要再启动已经完成计算的节点，避免了最初预想方式所产生的一个困难点。

通过上面的方式，直到所有的数据冗余文件均被删除，表示所有正确工作的节点计算完成，并且故障节点的任务被 host 接替计算完成，至此 reduce 计算的并行部分真正完成。

4.9.3 merge

对于 merge 函数，当 device 从 host 处获得要计算的数据时，host 会在 pool 中留下冗余数据文件，文件名同样包含 device 所运行的计算进程的进程号 client_pid，用来区分不同的计算节点，文件的内容存储了获得的数据和其他信息，比如 ip 地址等，与 map 不同的是 merge 的数据是两个数，而 map 的数据是一个数。

当 host 将任务分发完成后，pool 中至少有一个刚刚发出的任务，可能还会有许多计算还没有完成，pool 中还留有很多上述的冗余数据文件，此时仍然会有一些 device 上交计算结果，并试图获取新的计算任务，此时 host 并不终止该 device 的计算，而是从 pool 的冗余文件中挑选一个其它 device 还没有算完的任务分配给它。以这种方式分配出去的任务仍然要建立冗余数据文件，用来防止该 device 出现故障。

此时剩余没有算完的数据可能有很多 device 在同时计算，他们之中也许有出现故障的节点，也许只是因为计算所需时间较长，计算还没有结束。但终究有一个时刻，最先完成计算的节点会向 host 提交数据，此时，host 在清除该节点的冗余数据的同时，要检查 pool 中所有的冗余文件，凡是找到和该数据完全相同的文件，即表示获得了同样得到该任务的其它节点的信息，然后使用和 map 相同的方法判断故障，要把故障节点的 IP 地址从 device_ip 文件中删除，使其不影响后续的计算。

冗余文件全部检查完毕后，表示这个任务成功完成，可将这些冗余文件全部删除，当这些 device 再回来提交任务的时候 host 发现它的数据冗余文件已经被删除，就知道该 device 的任务已被其它 device 完成，于是放弃掉它提交的结果，并分给它新的任务，如果已没有任何新的任务，就向它发送计算结束的信息。当任务分发完毕，并且 pool 中没有任何任务数据的冗余文件时，就表示此次 merge 计算的并行部分真正完成。

4.10 并行化执行流程图

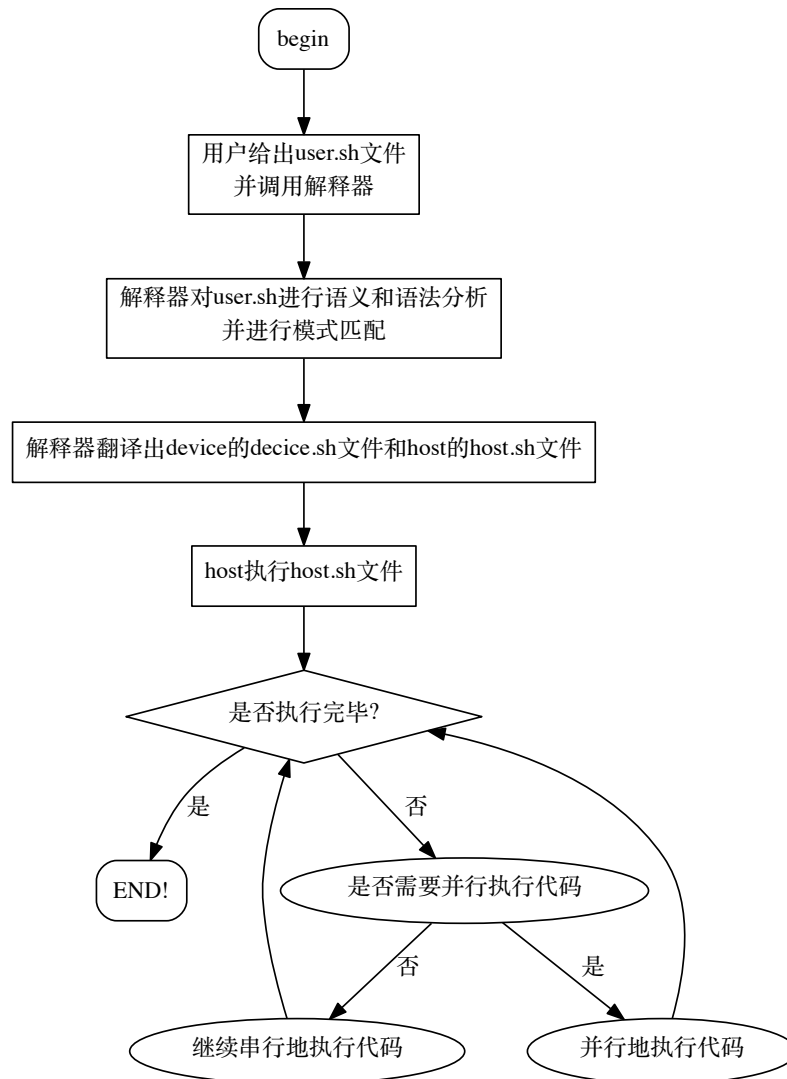


Figure 4.4: 总体执行流程图

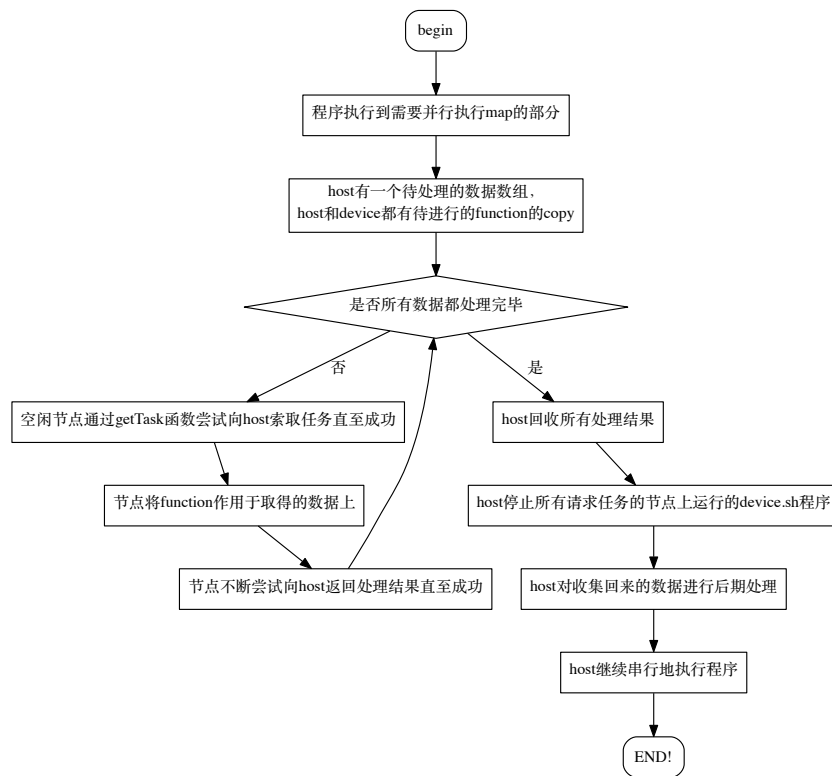


Figure 4.5: map 执行流程图

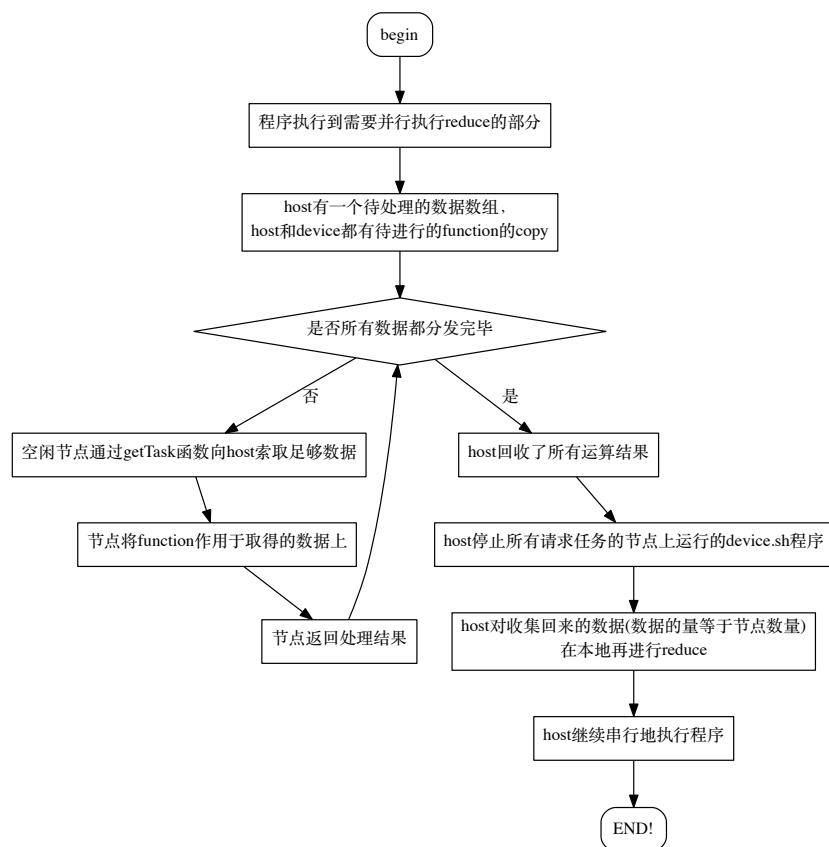


Figure 4.6: reduce 执行流程图

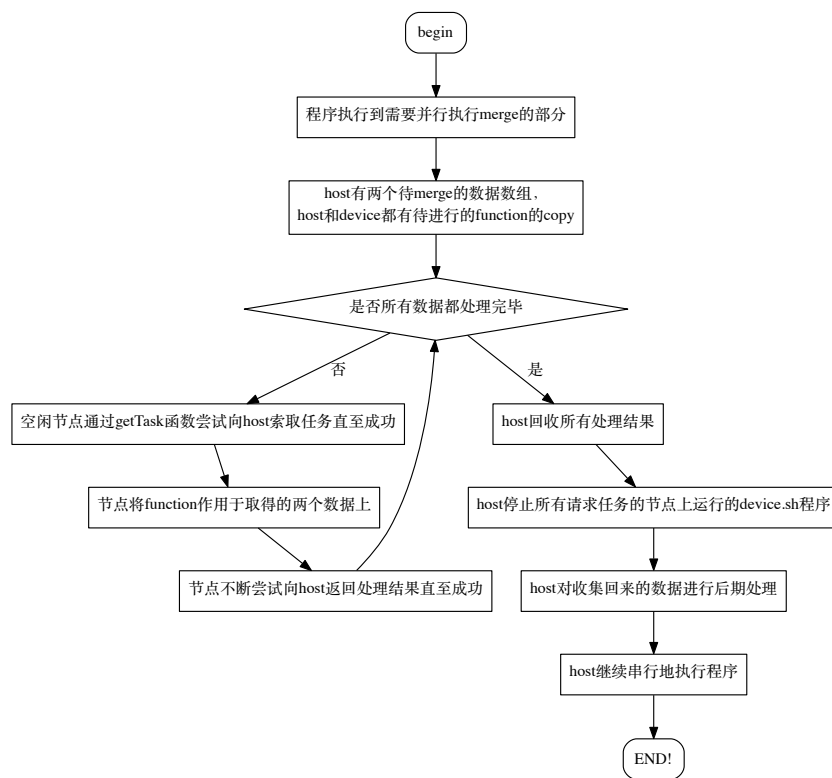


Figure 4.7: merge 执行流程图

4.11 解释器的总体架构

解释器的总体架构图如下:

interpreter.cpp					
userFileParser.h			userFileParser.cpp		
deviceCodeGenerator.h	deviceCodeGenerator.cpp		hostCodeGenerator.h	hostCodeGenerator.cpp	
map.h	map.cpp	merge.h	merge.cpp	reduce.h	reduce.cpp
configParser.h	configParser.cpp	map.template	reduce.template	merge.template	
templateConverter.h			templateConverter.cpp		

Figure 4.8: 解释器的总体架构图

其中 interpreter.cpp 为顶层模块, 为解释器的顶层架构代码.

第二层的 userFileParser.h 和 userFileParser.cpp 为 user.sh 的解析器, 主要是对 user.sh 进行词法分析和语法分析提取出有关信息, 便于第三层生成 device 和 host 代码.

第三层的 deviceCodeGenerator.h 与 deviceCodeGenerator.cpp 为 device 代码生成器. hostCodeGenerator.h 与 hostCodeGenerator.cpp 为 host 代码生成器. 它们使用第二层所得的信息分析并生成串行化部分代码, 同时对并行部分进行类型判断并调用第四层的相应模块产生并行部分代码.

第四层为并行模块代码的生成器. map.h 与 map.cpp 负责生成 map 部分的并行代码, merge.h 与 merge.cpp 负责生成 merge 部分的并行代码, reduce.h 与 reduce.cpp 负责生成 reduce 部分的并行代码.

第五层的 configParser.h 与 configParser.cpp 为 template 文件解析器. 它会读入 template 文件并将其分块解析并缓存供第六层模块使用. map.template, reduce.template, merge.template 为相应并行模块的 template 供 configParser 使用, 他们是根据上文的 3 个函数并行化原理写成的模板文件, 便于下层进行模式匹配.

第六层的 templateConverter.h 与 templateConverter.cpp 接受第五层的缓存信息, 并结合上层传递下来的语法分析信息进行模式匹配, 生成并行部分代码.

5 并行计算框架的安装与用法

5.1 安装方法

由于需要较多的源文件混合编译, 我们采取使用 GNU automake 和 autoconf 套装进行自动化编译. 用户需先安装 ≥ 1.11 的 automake 以及 ≥ 2.13 的 autoconf. 然后进入源文件夹执行 `./configure` 进行自动配置. 然后再通过 `make`, 完成自动编译. 最终通过 `make install` 将程序安装在 `/usr/bin` 中. 若想清理编译的中间文件可执行 `make clean`.

5.2 配置 device_ip

在运行并行程序之前, 我们需要配置 device 的 IP 地址. 配置十分简单, 只需在源文件夹里创建 `device_ip` 这一文件, 首先填 `127.0.0.1` 代表 localhost 地址即 host 地址. 接着再填入各个 device 的 IP 地址, 注意各个地址之间需空一格.

示例:

```
127.0.0.1 192.168.0.1 192.168.0.2
```

5.3 user.sh 的编写

用户首先需编写 `user.sh`, 然后经解释器翻译成并行化代码再执行. 所以 `user.sh` 的编写非常重要.

`user.sh` 的语法格式和 Inferno Shell 及其相似. 其中串行部分格式和 Inferno Shell 完全相同. 并行部分格式如下:

```
@parallel_begin
parallel code blocks .....
@parallel_end
```

其中 `parallel code blocks` 可以是 `map`, `reduce` 和 `merge`. 一个完整的 `user.sh` 示例如下:

```
#!/dis/sh -n
```

```

load std mpexpr

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

subfn combination {
    result = ${expr ${factorial $1} ${factorial
        $2} ${factorial ${expr $1 $2 -}} '*'
        '/'}
    if {ntest ${expr $1 $2 '<'}} {
        result = 0;
    }
}

subfn catalan {
    result = ${combination $1 ${expr $1 2
        '/' }}
}

subfn funct {
    result = ${expr $1 $2 +}
}

subfn mul {
    result = ${expr $1 $2 '*'}
}

```

```
}

array1 = 120 121 122 123 124 125 126 127 128
        129 130

@parallel_begin
map array1 calculate
@parallel_end

echo $array1

array2 = 1 2 3 4 5 6 7 8 9 10 11

@parallel_begin
map array2 catalan
@parallel_end

echo $array2

@parallel_begin
merge array1 array2 funct array3
@parallel_end

echo $array3

@parallel_begin
reduce array3 mul acc
@parallel_end

echo $acc
```

5.4 执行一个并行化程序

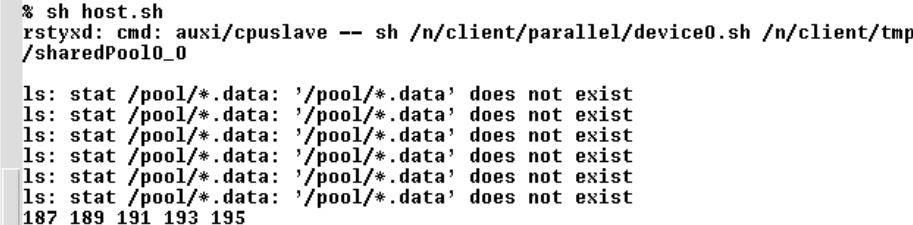
只需进入 Inferno 终端, 并进入工作目录 `/usr/inferno/parallel`. 在此目录下编写 `user.sh`. 之后执行 `sh finish.sh` 即可完成对 `user.sh` 的翻译生成 `device` 和 `host` 代码.



```
Shell /parallel
% cd parallel
% ls
device_ip
finish.sh
map.template
merge.template
user.sh
% sh finish.sh
% ls
device0.sh
device_ip
finish.sh
host.sh
map.template
merge.template
user.sh
%
% |
```

Figure 5.1: 解释器解释 `user.sh` 生成 `device` 和 `host` 代码

之后再执行 `sh host.sh` 即可由 `host` 调度各个 `device` 进行并行化执行, 得到结果. 整个过程非常方便.



```
% sh host.sh
rstyxd: cmd: auxi/cpuslave -- sh /n/client/parallel/device0.sh /n/client/tmp
/sharedPool0_0

ls: stat /pool/*.data: '/pool/*.data' does not exist
ls: stat /pool/*.data: '/pool/*.data' does not exist
ls: stat /pool/*.data: '/pool/*.data' does not exist
ls: stat /pool/*.data: '/pool/*.data' does not exist
ls: stat /pool/*.data: '/pool/*.data' does not exist
ls: stat /pool/*.data: '/pool/*.data' does not exist
187 189 191 193 195
```

Figure 5.2: 执行 `host.sh` 进行计算

6 成果与不足

6.1 向官方提交了 2 个 bug 与 1 个 patch

6.1.1 arm 架构下的编译 bug

发现 arm 架构下存在编译的 bug. 第一时间通知官方, 并被标记位 critical 级别. 不久在和官方的共同努力下成功修复此 bug.

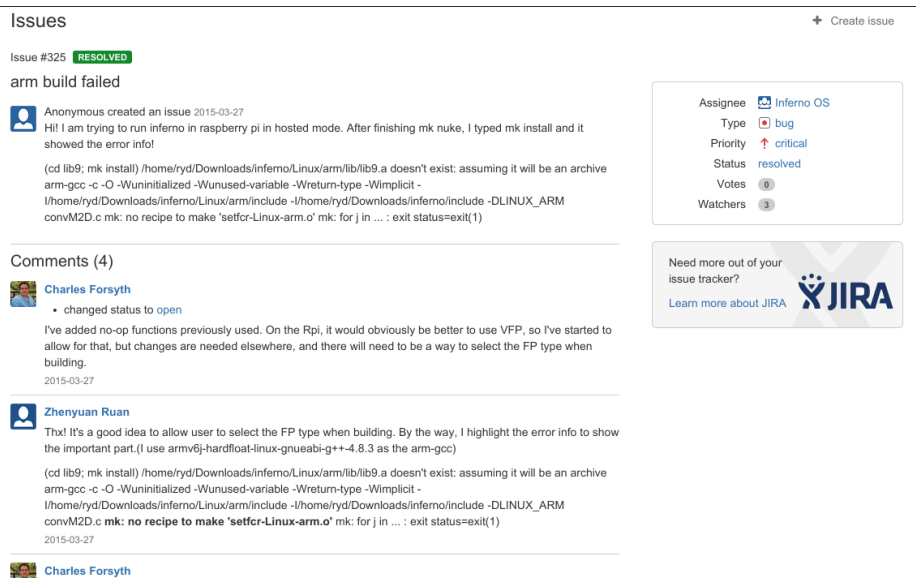


Figure 6.1: arm 架构下的编译 bug

6.1.2 arm 架构下的网络接口 bug

发现了 arm 架构下网络接口存在问题, 并第一时间通知官方. 被官方标记未 major 级别.

6.1.3 有关 x86 编译问题的 patch


发现了 x86 下存在编译问题, 第一时间修复并通知官方. 官方告知不久之前他们也以修复, 但还未来得及更新源码.

Issues

Issue #326

NEW

arm listen failed (because IPv6 support is optional in kernels)




Zhenyuan Ruan

created an issue 2015-03-27


It's seems that the protocol in arm doesn't work normally.

```
pi@raspberrypi ~$ ./Linux/arm/bin/emu fs: fsqid: top-bit dev: 0xb302 ; svc/auth Key: listen: failed to announce on 'tcp!linfo!gin': Address family not supported by protocol listen: failed to announce on 'tcp!linfo!fkey': Address family not supported by protocol listen: failed to announce on 'tcp!linfo!signer': Address family not supported by protocol listen: failed to announce on 'tcp!linfo!signer': Address family not supported by protocol
```


Assignee

 Inferno OS


Type

 bug


Priority

 major


Status

 new


Votes

 0

Watchers

 1

Comments (5)




Charles Forsyth

There are two things there. The "top-bit dev" is something else that needs looking at, although I know roughly what that is (finally having to address some very early laziness!). It might not matter much at the moment.

The main problem is the "address family". I wonder if it's because it's trying to use IPv6 interfaces but your kernel hasn't got them configured. No wonder adoption of ipv6 is slow. I don't know why Linux doesn't simply allow ipv6 addressing families but restrict the targets to ipv4 space. Anyway, having looked at the emu code I suspect that's the cause.

If so, change the line in emu/Linux/emu from ip ip!f6-posix ipaux to ip ip!f-posix ipaux and remake emu.


2015-03-27



Charles Forsyth

The top-bit dev message is to let someone know that it's happened. It then compensates. I've just changed dev!f6-posix.c to suppress the message since it's no longer needed.

2015-03-27



Charles Forsyth


It should probably work out at run-time whether IPv6 is available and use the older calls if not.

+

Create issue

Need more out of your issue tracker?

Learn more about JIRA



Need more out of your issue tracker?

[Learn more about JIRA](#)


The JIRA logo, featuring a stylized 'J' icon followed by the word 'JIRA' in a bold, sans-serif font.

Issues

Issue #329

NEW

A patch for x64



Zhenyuan Ruan

created an issue 2015-03-28

If you're building on Linux/amd64 system, you will require to import this patch to fix a compile error - floating point exception.


```

-- a/Linux386/include/fpuctrl.h Sat Mar 08 13:03:33 2015 +0000
+++ b/Linux386/include/fpuctrl.h Sat Mar 28 02:16:23 2015 +0800
<at> <at> -6,12 +6,12 <at> <at>


static void
setfcr(julong fcr)
{
    _asm_ ( "xorb $0x3f, %%eax\n\t"
+ _asm_ volatile( "xorb $0x3f, %%eax\n\t"
"pushw %%eax\n\t"
"twaihr\n\t"
"fdcw (%%eax)\n\t"
"popw %%eax\n\t"
+ : /* no output */
+ : "a" (fcr)
+ : "a" (fcr)
);
}

```


Assignee

 Inferno OS

Type

 bug

Priority

 major

Status

new


Votes

0


Watchers

3

Need more out of your issue tracker?

[Learn more about JIRA](#)


Comments (2)




Zhenyuan Ruan

REPORTER

- edited description

2015-03-28



Charles Forsyth

this interface was replaced on 24 March. I did away with fpuctrl.h (it's now an empty file) and the floating-point control operations are now accessed from assembly language instead of having to deal with the extra layer of obscurity in asm. They aren't on a critical path so the inline assembly isn't important.

2015-04-02

6.2 日志功能方便调试

在程序的执行过程中，会在 pool 中产生 log 日志文件，包括 host 的日志 host.log 和每个 device 的日志，方便查看中间结果和调试程序。

host.log 的内容包括来提交计算结果的进程 pid 和提交结果之后的当前计算结果。如图：

```
root@Linux:/usr/inferno/pool# cat host.log
from 43 current answer is 1 116
from 43 current answer is 1 116 2 118
from 43 current answer is 1 116 2 118 3 120
from 2173398 current answer is 1 116 2 118 3 120 0 114
from 43 current answer is 1 116 2 118 3 120 0 114 4 122
from 43 current answer is 1 116 2 118 3 120 0 114 4 122 6 126
from 43 current answer is 1 116 2 118 3 120 0 114 4 122 6 126 7 128
from 2173398 current answer is 1 116 2 118 3 120 0 114 4 122 6 126 7 128 9 132
from 2173398 current answer is 1 116 2 118 3 120 0 114 4 122 6 126 7 128 9 132 10 134
from 2173398 current answer is 1 116 2 118 3 120 0 114 4 122 6 126 7 128 9 132 10 134 11 136
from 2173398 current answer is 1 116 2 118 3 120 0 114 4 122 6 126 7 128 9 132 10 134 11 136 12 138
from 2173398 current answer is 1 116 2 118 3 120 0 114 4 122 6 126 7 128 9 132 10 134 11 136 12 138 8 130
from 2193748 current answer is 0 121
from 2193748 current answer is 0 121 1 123
from 2193748 current answer is 0 121 1 123 2 125
from 2193748 current answer is 0 121 1 123 2 125 3 127
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131 6 133
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131 6 133 7 135
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131 6 133 7 135 8 137
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131 6 133 7 135 8 137 9 139
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131 6 133 7 135 8 137 9 139 10 141
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131 6 133 7 135 8 137 9 139 10 141 11 143
from 2193748 current answer is 0 121 1 123 2 125 3 127 4 129 5 131 6 133 7 135 8 137 9 139 10 141 11 143 12 145
```

Figure 6.4: host 的日志

device 的日志包括该节点获得的数据和将要提交给 host 的计算结果。如图：


```
root@Linux:/usr/inferno/pool# cat Linux.log
Linux is pending
Linux fetched 0 120 1
0 120 1 put_ans : 0 114
Linux fetched 5 125 6
5 125 6 put_ans : 5 124
Linux fetched 9 129 10
9 129 10 put_ans : 9 132
Linux fetched 10 130 11
10 130 11 put_ans : 10 134
Linux fetched 11 131 12
11 131 12 put_ans : 11 136
Linux fetched 12 132 13
12 132 13 put_ans : 12 138
Linux fetched 8 128 9
8 128 9 put_ans : 8 130
2173398 Linux is finished
```

Figure 6.5: device 的日志

6.3 故障修复

当 macbook 节点计算到编号为 7 的数据时，出现故障，对应黄色边框。

当 yangzhi-GE70-0NC-GE70-0ND 节点取得编号为 11 的数据时，出现故障，对应红色边框。

最后，当 Linux 节点处理完分配的任务后，重新计算之前出现故障的节点的数据，保证了最后结果的正确性。

```
root@Linux:/usr/inferno/pool# cat macbook.log
macbook is pending
macbook fetched 2 133
2 133 put_ans : 2 126
macbook fetched 4 135
4 135 put_ans : 4 128
macbook fetched 7 138
```

Figure 6.6: macbook 节点取编号为 7 的数据时出现故障

```
root@Linux:/usr/inferno/pool# cat yangzhi-GE70-0NC-GE70-0ND.log
yangzhi-GE70-0NC-GE70-0ND is pending
yangzhi-GE70-0NC-GE70-0ND fetched 1 132
1 132 put_ans : 1 125
yangzhi-GE70-0NC-GE70-0ND fetched 3 134
3 134 put_ans : 3 127
yangzhi-GE70-0NC-GE70-0ND fetched 6 137
6 137 put_ans : 6 130
yangzhi-GE70-0NC-GE70-0ND fetched 8 139
8 139 put_ans : 8 132
yangzhi-GE70-0NC-GE70-0ND fetched 9 140
9 140 put_ans : 9 133
yangzhi-GE70-0NC-GE70-0ND fetched 11 142
```

Figure 6.7: yangzhi 节点取编号为 11 的数据时出现故障

6.4 跨平台性

由于 Linux, Unix 以及 Windows 均支持 hosted 模式的 Inferno, 故此框架具有天然的跨软平台型. 用户可以在不更改当前操作系统环境的条件下使用这套并行计算框架. 这个使用者的部署 (deploy) 带来了极大的方便, 只

```
root@Linux:/usr/inferno/pool# cat Linux.log
Linux is pending
Linux fetched 0 131
0 131 put_ans : 0 124
Linux fetched 5 136
5 136 put_ans : 5 129
Linux fetched 10 141
10 141 put_ans : 10 134
Linux fetched 12 143
12 143 put_ans : 12 136
Linux fetched 13 144
13 144 put_ans : 13 137
Linux fetched 14 145
14 145 put_ans : 14 138
Linux fetched 7 138
7 138 put_ans : 7 131
Linux fetched 11 142
11 142 put_ans : 11 135
312029 Linux is finished
```

Figure 6.8: Linux 节点在完成自身任务后重新计算因故障未被计算的数据

需本地安装 Inferno 并编译安装解释器即可使用。

同时由于 Inferno 自身支持 x86, amd64, arm, spark, mips, powerPC 等多种指令集体系结构, 故此框架具有天然的跨硬平台型。

综上, 此并行计算框架可以在各种软环境, 硬平台下执行, 给使用者带来了极大的方便。

6.5 高度可扩展性与可配置性

现在我们的并行计算框架仅有 map, reduce 与 merge 函数。然而介于解释器优良的架构, 扩展其他函数变得非常的容易。由解释器的架构图可知, 倘若我们想再往框架中添加 fun1 这个函数, 我们只需参照 map.h 与 map.cpp 的写法往第四层添加 fun1.h 和 fun1.cpp。同时编写 fun1 对应的 template, 扩展第五层即可。整个扩展的过程非常简单, 用户可以在不了解解释器底层

interpreter.cpp					
userFileParser.h			userFileParser.cpp		
deviceCodeGenerator.h	deviceCodeGenerator.cpp		hostCodeGenerator.h	hostCodeGenerator.cpp	
map.h	map.cpp	merge.h	merge.cpp	reduce.h	reduce.cpp
configParser.h	configParser.cpp	map.template		reduce.template	merge.template
templateConverter.h			templateConverter.cpp		

Figure 6.9: 解释器的总体架构图

工作原理的情况下轻松扩展解释器。

同时由于我们采用了 template 的机制, 将解释规则和解释内容分隔开来. 解释器中制订了一套规则, 而具体解释成什么由 template 来定. 解释器在读取 user.sh 进行语法分析之后, 再读入 template 进行模式匹配完成翻译. 若想优化翻译出的代码只需修改 template 即可, 而解释器的内容可以一行不改. 我们将解释器与 template 分离使其具有了高度的可配置型.

6.6 benchmark

6.6.1 各个设备的配置

我们使用了树莓派, 以及组员四台电脑合计 5 台设备进行 benchmark 测试. 其中五台设备的具体信息如下:

树莓派 型号: Raspberry Pi 2

CPU : ARM Cortex-A7 @ 900MHz

RAM : LPDDR2 SDRAM 1GB

Disk : kingston 16GB SD Card.

刘旭彤的设备 型号: 2014 款 MacBook Air 13' 低配

CPU : Intel Core i5-4260U @ 1.4 GHz

RAM : DDR3 4 GB 1600 MHz

Disk : Samsung SSD 128GB.

阮震元的设备 型号: 2013 款 MacBook Air 13' 高配

CPU : Intel Core i5-4250U @ 1.3 GHz

RAM : DDR3 4 GB 1600 MHz

Disk : Samsung SSD 256GB.

解宇飞的设备 型号: Terrans Force x511

CPU : Intel Core i7-4700MQ @ 2.40GHz

RAM : DDR3 16G 1600 MHz

Disk : 西部数据 1TB 7200 转

杨智的设备 型号: MSI G1790

CPU : Intel Core i7 3630QM @ 2.40GHz

RAM : DDR3 8G 1600 MHz

Disk : 日立 750GB 7200 转

6.6.2 benchmark 内容

我们采用检验给定表中素数个数来作为我们的压力测试 (benchmark). 首先为了保证有一定计算压力, 我们采用复杂度较高的试除法作为素数检验算法, 而不是基于随机性的素数测试. 试除法的检验数 x 是不是素数的过程如下:

1. 从 2 到根号 x (取下整) 枚举 i , 若枚举完毕则跳 (3).
2. 检验 x 是否整除 i , 若整除则 x 是合数, 否则返回 (1).
3. x 是素数.

由于不同设备的性能, 我们先在每个设备上运行串行版本的 benchmark 进行各个设备的性能基准测试. 之后通过我们的并行框架将各个设备互联运行并行版本的 benchmark.

我们选用了百余个大素数 (十亿级别) 掺杂着少量合数组成名为 input 的表. 用试除法编写了名为 isprime 的 subfn. 再时编写了名为 add 的二元和函数.

串行版本的 **benchmark** 枚举 input 中的每个数字 x, 再用试除法判断 x 是不是素数, 如果是使答案加 1. 最终输出答案. 具体代码如下:

```
#!/dis/sh -n
load std mpexpr

subfn isprime {
    list = ${expr 2 ${sqrt $1} seq}
    ok = 0
    for i in $list {
        if {~ ${expr $1 $i '%' } 0} {
            ok = 1
            raise break
        }
    }
    if {~ $ok 1} {
        result = 0
    } {
        result = 1
    }
}

subfn add {
    result = ${expr $1 $2 +}
}

input = 2038074750 2038074751 2038074752
        2038075231 2038075751 2038076267 2038074761
        2038075271 2038075777 2038076281 2038074769
        2038075289 2038075783 2038076303 2038074793
        2038075301 2038075817 2038076311 2038074803
        2038075307 2038075883 2038076317 2038074817
        2038075339 2038075909 2038076323 2038074853
        2038075349 2038075913 2038076329 2038074923
```

```

2038075363 2038075933 2038076351 2038074931
2038075373 2038075943 2038076357 2038074947
2038075387 2038075967 2038076423 2038074949
2038075397 2038076003 2038076449 2038075001
2038075423 2038076057 2038076473 2038075019
2038075433 2038076083 2038076479 2038075037
2038075463 2038076099 2038076489 2038075049
2038075469 2038076101 2038076507 2038075069
2038075511 2038076111 2038076539 2038075099
2038075531 2038076113 2038076543 2038075121
2038075547 2038076119 2038076627 2038075163
2038075579 2038076129 2038076639 2038075183
2038075597 2038076137 2038076653 2038075187
2038075619 2038076171 2038076717 2038075189
2038075639 2038076191 2038076723 2038075199
2038075643 2038076213 2038076737 2038075219
2038075649 2038076221 2038076767 2038075229
2038075691 2038076233 2038076783

ans = 0

for i in $input {
    if {~ ${isprime $i} 1}
        ans = ${add $ans 1}
}

echo "There is $ans primes in the input list"

```

并行版本的 benchmark 首先通过 `map input isprime` 将 `input` 映射为只含 0/1 的表, 素数被映射为 1, 合数被映射为 0. 最后通过 `reduce input add ans` 得到素数的总个数. 具体代码如下:

```
#!/dis/sh -n
```

```

load std mpexpr

subfn isprime {
    list = ${expr 2 ${sqrt $1} seq}
    ok = 0
    for i in $list {
        if {~ ${expr $1 $i '%' } 0} {
            ok = 1
            raise break
        }
    }
    if {~ $ok 1} {
        result = 0
    } {
        result = 1
    }
}

subfn add {
    result = ${expr $1 $2 +}
}

input = 2038074750 2038074751 2038074752
2038075231 2038075751 2038076267 2038074761
2038075271 2038075777 2038076281 2038074769
2038075289 2038075783 2038076303 2038074793
2038075301 2038075817 2038076311 2038074803
2038075307 2038075883 2038076317 2038074817
2038075339 2038075909 2038076323 2038074853
2038075349 2038075913 2038076329 2038074923
2038075363 2038075933 2038076351 2038074931
2038075373 2038075943 2038076357 2038074947
2038075387 2038075967 2038076423 2038074949

```



```
2038075397 2038076003 2038076449 2038075001
2038075423 2038076057 2038076473 2038075019
2038075433 2038076083 2038076479 2038075037
2038075463 2038076099 2038076489 2038075049
2038075469 2038076101 2038076507 2038075069
2038075511 2038076111 2038076539 2038075099
2038075531 2038076113 2038076543 2038075121
2038075547 2038076119 2038076627 2038075163
2038075579 2038076129 2038076639 2038075183
2038075597 2038076137 2038076653 2038075187
2038075619 2038076171 2038076717 2038075189
2038075639 2038076191 2038076723 2038075199
2038075643 2038076213 2038076737 2038075219
2038075649 2038076221 2038076767 2038075229
2038075691 2038076233 2038076783

@parallel_begin
map input isprime
@parallel_end

@parallel_begin
reduce input add ans
@parallel_end

echo "There is $ans primes in the input list"
```

6.6.3 性能基准测试结果

五台设备均运行串行版本的 benchmark 进行性能基准测试. 如图所示, 树莓派跑完 benchmark 的时间远超于其他设备, 性能较差. 刘旭彤的设备速度约比阮震元设备慢一倍. 杨智的设备比阮震元的更快一些, 解宇飞的设备速度最快.

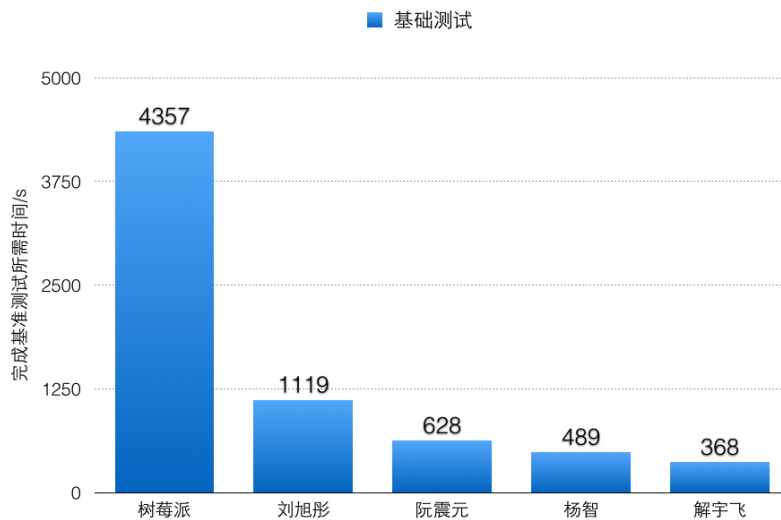


Figure 6.10: 五台设备性能基准测试结果

6.6.4 使用并行计算框架进行并行 benchmark 测试

所得结果如图所示. 首先用树莓派使用并行计算框架的进行单节点测试 (即虽然使用并行计算框架, 但仅有树莓派这一个节点参与计算) 所得时间为 4493s, 与树莓派运行串行代码相比略慢了一些. 这是由并行计算框架中通讯以及同步耗时所带来的.

接着, 树莓派加上刘旭彤设备进行双节点并行计算. 仅需 937s 就跑完了整个 benchmark, 速度大幅提升.

加上阮震元的设备进行 3 节点并行计算仅需要 307s.

加上解宇飞的设备进行 4 节点并行计算仅需要 175s.

加上杨智的设备进行 5 节点并行计算仅需要 110s.

由以上结果我们可以计算出实际加速比. 同时我们可以根据基准测试推算出加速比的理论值. 将两者绘于一张图上进行比较. 由图可知, 我们的并行计算框架性能较好, 实际加速比值接近理论加速比值.

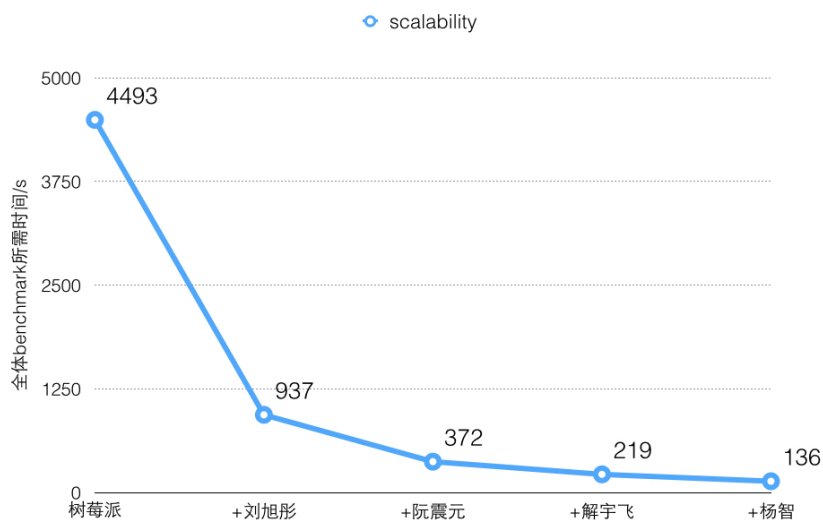


Figure 6.11: 使用并行计算框架进行并行 benchmark 测试

6.7 与其他并行计算框架的对比

6.7.1 异构性

我们最终实现的并行计算框架异构性较好。

节点 A 运算完毕花了 4357 秒，节点 B 花了 1119 秒，节点 C 花了 628 秒，节点 D 花了 489 秒，节点 E 花了 368 秒。但当这四个节点组成集群进行同一项测试时，只花了 119 秒就完成了。由此可见，运算较慢的节点 A 非但没有拖累整个集群的整体性能，反而提高了集群的运算速度。

另一方面，该并行计算框架不要求各节点的框架一致、不要求操作系统一致（Inferno 可以安装在另一个操作系统上）、不要求硬件相似或者相同，只要节点能够运行 Inferno，我们就能把它加入到集群中，提高集群的运算能力。

与 MPI 相比，我们在异构性这方面占了极大的优势。

一方面，使用 MPI 进行并行计算时，为了保证各个进程间的同步，有时候不得不调用名为 MPI_BARRIER 的函数，通过这种方式阻止调用直

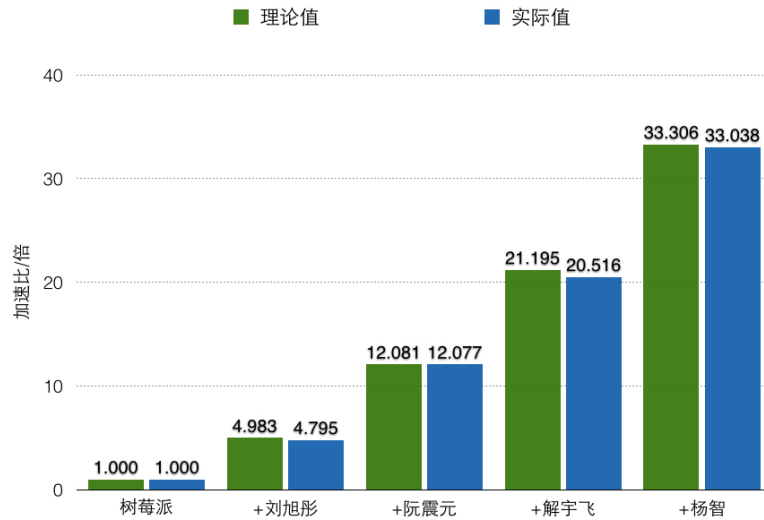


Figure 6.12: 实测所得加速比与理论计算所得加速比 (较单节点树莓派)

到 communicator 中所有进程已经完成调用，实现同步。这种方式就有可能导致运算速度慢的节点拖累其它节点的运算。

又如在 MPI，以一种动态负载均衡算法为例，我们需要考虑 cpu 队列长度、cpu 利用率、可用内存大小、上下文切换速度等因素来评价各节点负载状态，要决定如何收集系统当前负载的信息、是否进行任务迁移以及向何处迁移、如何进行迁移，一旦处理不好，就容易出现硬件水平差的节点影响集群整体速度的情况。

另一方面，MPI 还要求集群中的各个节点硬件相似或者相同，要求各节点是同一框架，异构性不好。

6.7.2 部署简单

我们的并行计算框架 下载之后只需 `./configure`, `/make`, `/make install` 三条命令即可安装。使用方法也非常方便，只需配置 `device_ip`，然后运行 `finish.sh` 即可生成并行化可执行代码。由于我们基于 `inferno` 分布式操作系统，所以我们生成的 `host` 和 `device` 代码只需存在 `host` 上即可，不需要把

可执行的程序复制到每台机器上。非常易于使用。

CUDA 为使用 CUDA 我们需要安装 3 个软件包，分别是：CUDA Driver, CUDA Toolkit, CUDA SDK。然后还要配置系统环境变量和编译环境，代码要经过专有编译器 `nvcc` 编译之后才可使用。使用起来较为麻烦。

MPI 首先要在每台机器上安装 MPI 套件，例如 Intel MPI、OpenMPI 等套装，有时在安装时要进行繁琐的编译参数配置，安装后要配置系统环境变量和编译环境，多台节点同时运行时要至少注册一个 `mpi` 账号，然后创建 JOB 目录，每台机器上必须有一个路径相同的目录，再将可执行文件复制到其它所有的机器上。使用起来非常麻烦。

6.7.3 编程简单

我们小组并行框架相对于主流并行框架拥有更强的易用性。下面我们与主流并行框架：Open Mp,CUDA,OpenCL,MPI 一一进行比较：

我们的并行计算框架 我们小组的并行框架易用性是极强的，用户所需要做的只是按要求将函数和数组作为参数传入，在想要并行的部分之前加上 `@parallel_begin`，在结束的地方加上 `@parallel_end`，中间即为要并行的代码。举个例子，通过 `map` 操作将表 `array1` 中每个元素加 1：

```
subfn calculate {
    result = ${expr $1 1 +}
}
array1 = 120 121 122 123 124 125 126 127 128
        129 130
@parallel_begin
map array1 calculate
@parallel_end
```

由此可见，即使是刚使用本框架的用户也可以极快上手。那些繁杂的并行化部分都是由解释器解决的，对于用户是透明的。

OpenMP OpenMP 的编程还是相对简单的，我们以一个简单的 `for` 循环作为例子：

```

#include <iostream>

using namespace std;

int main()
{
#pragma omp parallel for
    for (int i=0; i<10; i++)
    {
        cout << i;
    }
    return 0;
}

```

由上面的例子可以看出：要想将你的程序变为一个 OpenMp 的并程序，只需加上一行代码 `#pragma omp parallel for`. 简单的来说，就是将你想要并行化的代码加上 OpenMp 的并行标示 `#pragma omp parallel` 就可以完成并行化 (线程级别并行), 具体的工作将交于编译器完成, 也具有较强的易用性.

CUDA CUDA 允许用户进行线程级的并行。用户需要先了解：CUDA 的每一个线程都有其线程 ID，线程的 ID 信息由变量 `threadIdx` 给出。`threadIdx` 是 CUDA C 语言的内置变量，通常它用一个三维数组来表示。由此表示一维、二维和三维线程块 (thread block)。这样，无论是数组、矩阵还是体积的计算，都可以使用 CUDA 进行运算。举两个 NxN 矩阵相加的 CUDA 实现为例子：

```

// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

```

```

int main() {
    ...
    // Kernel invocation with one block of N *
    N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B
        , C);
    ...
}

```

可以看出用户需要先定义一种被称为内核函数（Kernel Functions）的 C 函数。内核函数使用关键字 `__global__` 来声明，运行该函数的 CUDA 线程数则通过 `<<<...>>>` 执行配置语法来设置。每一个执行内核函数的线程都由一个唯一的线程 ID，这一 ID 可以通过在内核函数中访问 `threadIdx` 变量来得到。所以，用户要先定义内核函数再进行线程分配，这样就与我们小组并行框架的易用性有一定差距。

OpenCL OpenCL 是通用的异构平台编程语言，为了兼顾不同设备，使用极为繁琐。将 OpenCL 编程步骤进行归纳，可得以下 13 步：

1. Discover and initialize the platforms. 调用两次 `clGetPlatformIDs` 函数，第一次获取可用的平台数量，第二次获取一个可用的平台。
2. Discover and initialize the devices. 调用两次 `clGetDeviceIDs` 函数，第一次获取可用的设备数量，第二次获取一个可用的设备。
3. Create a context(调用 `clCreateContext` 函数)。上下文 context 可能会管理多个设备 device。
4. Create a command queue(调用 `clCreateCommandQueue` 函数)。一个设备 device 对应一个 command queue。上下文 context 将命令发送到设备对应的 command queue，设备就可以执行命令队列里的命令。
5. Create device buffers(调用 `clCreateBuffer` 函数)。Buffer 中保存的是数据对象，就是设备执行程序需要的数据保存在其中。Buffer 由上下

文 conetxt 创建，这样上下文管理的多个设备就会共享 Buffer 中的数据。

6. Write host data to device buffers(调用 clEnqueueWriteBuffer 函数) .
7. Create and compile the program. 创建程序对象，程序对象就代表你的程序源文件或者二进制代码数据。
8. Create the kernel(调用 clCreateKernel 函数) . 根据你的程序对象，生成 kernel 对象，表示设备程序的入口。
9. Set the kernel arguments(调用 clSetKernelArg 函数) .
10. Configure the work-item structure(设置 worksize) . 配置 work-item 的组织形式（维数，group 组成等） .
11. Enqueue the kernel for execution(调用 clEnqueueNDRangeKernel 函数) . 将 kernel 对象，以及 work-item 参数放入命令队列中进行执行。
12. Read the output buffer back to the host(调用 clEnqueueReadBuffer 函数) .
13. Release OpenCL resources（至此结束整个运行过程） .

由此可见：OpenCL 中的核函数必须单列一个文件。OpenCL 的编程一般步骤就是上面的 13 步，太长了，以至于要想做个向量加法都是那么困难。下面代码是一个例子，该函数是一个映射函数，但竟然要高达 90 余行的代码量，对于用户的易用性相当差。

```
// 核函数文件 , HelloWorld_Kernel.cl
__kernel void helloworld(__global double* in ,
    __global double* out)
{
    int num = get_global_id(0);
    out[num] = in[num] / 2.4 *(in[num]/6) ;
}

// 主函数文件 , HelloWorld.cpp
//For clarity ,error checking has been omitted.
```



```

#include <CL/cl.h>
#include "tool.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

int main(int argc, char* argv[])
{
    cl_int    status;
    /**Step 1: Getting platforms and choose an
        available one(first).*/
    cl_platform_id platform;
    getPlatform(platform);

    /**Step 2:Query the platform and choose the
        first GPU device if has one.*/
    cl_device_id *devices=getCl_device_id(
        platform);

    /**Step 3: Create context.*/
    cl_context context = clCreateContext(NULL
        ,1, devices ,NULL,NULL,NULL);

    /**Step 4: Creating command queue associate
        with the context.*/
    cl_command_queue commandQueue =
        clCreateCommandQueue(context, devices
            [0], 0, NULL);

```

```

/**Step 5: Create program object */
const char *filename = "HelloWorld_Kernel.
    cl";
string sourceStr;
status = convertToString(filename,
    sourceStr);
const char *source = sourceStr.c_str();
size_t sourceSize [] = {strlen(source)};
cl_program program =
    clCreateProgramWithSource(context, 1, &
        source, sourceSize, NULL);

/**Step 6: Build program. */
status=clBuildProgram(program, 1,devices,
    NULL,NULL,NULL);

/**Step 7: Initial input,output for the
    host and create memory objects for the
    kernel*/
const int NUM=512000;
double* input = new double[NUM];

for(int i=0;i<NUM;i++)
    input[i]=i;

double* output = new double[NUM];

cl_mem inputBuffer = clCreateBuffer(context
    , CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
    (NUM) * sizeof(double),(void *) input,
    NULL);

```

MPI MPI 对于用户的要求也是很高的，MPI 有几个很重要的概念 rank, group, communicator, type, pack, spawn, window, 用户只有理解了这些概念才可能了解 MPI 框架的使用方法。而这几个概念是身份抽象的。比如说 group 这个概念，一台电脑可以属于多个 group，group 也可以随时随地的组合任意 group，然后利用 group 内，和 group 间的 communicator，可以很容易实现复杂科学计算的中间过程。例如奇数 rank 一个 group，偶数另一个 group，或者拓扑结构的 group，这样可以解决很多复杂问题，另外 MPI 还有一个默认的全局的 group，它就是 comm world。

当然理解了这些概念也还是远远不够的，用户还需要掌握 MPI 的数据结构和许多编程技巧。比如一个简单的计算矩阵和的程序：

```
#include "mpi.h"

#include <iostream>
#include <fstream>

int main(int argc, char* argv[])
{
    using namespace std;
    int rank;
    int size;
    MPI_Status Status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int MAXX=8;
    const int MAXY=3;
    const int MAXPROCESSOR=64;
    float Data[MAXX][MAXX];
    int ArraySize[2];
    int i, j, k;

    if (rank == 0)
```

```

{
    ifstream in("input.txt");
    in>>ArraySize[0]>>ArraySize[1];
    for (i=1; i<=ArraySize[0]; i++)
    {
        for (j=1; j<=ArraySize[1]; j++)
        {
            in>>Data[i][j];
        }
    }
}

MPI_Bcast(ArraySize,2,MPI_INT,0,
MPI_COMM_WORLD);
int AverageLineNumber,HeavyProcessorNumber,
MyLineNumber;
int CurrentLine,StartLine,SendSize;
float SendArray[MAXX*MAXY];

AverageLineNumber=ArraySize[0] / size;
HeavyProcessorNumber=ArraySize[0] % size;
if (rank < HeavyProcessorNumber)
{
    MyLineNumber=AverageLineNumber+1;
}
else
{
    MyLineNumber=AverageLineNumber;
}

if (rank == 0)
{
    CurrentLine=ArraySize[0];

```

```

for (i=size-1; i >= 0; i--)
{
    SendSize=0;

    if (i < HeavyProcessorNumber)
        StartLine=CurrentLine-
            AverageLineNumber;
    else
        StartLine=CurrentLine-
            AverageLineNumber+1;

    for (j=StartLine; j <= CurrentLine;
        j++)
        for (k=1; k <= ArraySize[1]; k
            ++)
            SendArray[SendSize++]=Data[
                j][k];

    if (i != 0)
        MPI_Send(SendArray, SendSize,
            MPI_FLOAT, i, 10,
            MPI_COMM_WORLD);
    CurrentLine=StartLine-1;
}
}
else
{
    MPI_Recv(SendArray, MyLineNumber*
        ArraySize[1],
            MPI_FLOAT, 0, 10,
            MPI_COMM_WORLD, &Status)
        ;
}

```

```

float *Sum=new( float );
*Sum=0;
for ( i=0; i < MyLineNumber*ArraySize[1]; i
    ++ )
    *Sum+=SendArray[ i ];
float AllSum[MAXPROCESSOR];
std::cout << "rank:" << rank << " cal sum
    is " << *Sum << std::endl;
MPI_Gather( Sum,1,MPI_FLOAT, AllSum,1,
    MPI_FLOAT,0,MPI_COMM_WORLD);

if (rank == 0)
{
    *Sum=0;
    for ( i=0; i < size; i++)
        *Sum+=AllSum[ i ];
    cout<<"The Sum of the Array is:"<<*Sum
        <<endl;
}

MPI_Finalize();
return 0;
}

```

由此可见，MPI 对于用户的要求很高。完全无法想象一个刚接触 MPI 的用户如何能写出 MPI 框架下并行的代码。MPI 的易用性也因此大打折扣，和我们小组相比有很大的差距。

6.7.4 支持故障恢复

MPI 标准并没有提供容错的策略，在 MPI-1 中任务和主机都是静态的，应用程序需以一个任务组的方式运行，当任务组中的一个任务或计算资源失败以后，整个应用程序都将失败，这是 MPI 标准对错误的处理方式。由

于 MPI 标准中目前还缺乏关于容错的接口与操作的明确定义，众多的 MPI 版本需要在软件实现时解决容错问题。目前有一些 MPI 版本可以实现检查点设置/回卷恢复的功能。

基于分布式系统和数据冗余的方法，我们的并行框架可以做到任何 device 出现故障不影响程序的执行和结果的正确性，当 host 出现故障才会中断程序执行。

Table 6.1: summary

	我们的框架	CUDA	MPI	OpenCL	OpenMP
异构性	good	good	bad	good	\
部署难度	easy	normal	normal	normal	easy
编程难度	easy	normal	hard	crazy	easy
故障恢复	support	\	unsupport	\	\

6.8 不足点

- 无法加速时间复杂度小于 $O(n^2)$ 的算法（或者函数），例如对 n 个数据求和。具体的原因是 host 节点将数据分发出去，由于各个节点计算速度不同，收回结果的函数必定是乱序的。我们小组对于乱序的结果必须要进行排序，也就是说要根据 pid 为关键字进行排序。由于 Inferno 系统的自身因素，排序算法只能达到 $O(n^2)$ 。所以本身时间复杂度小于 $O(n^2)$ 的无法进行加速。
- 由于我们小组使用的是 Inferno 的内嵌语言，类似于 JVM，是对程序是进行逐行解释运行的。这导致我们小组设计的框架速度受限。
- 我们小组使用了 host-device 平台模型。这就说明 host 一旦宕机整个系统就会崩溃，无法再进行故障修复。但是，经过我们查阅了大量的资料发现：大量并行框架也是采用此模型，如 CUDA、OpenCL。可以预见，这两种模型也会出现类似的问题。当然，对于 MPI，由于使用了广播与点对点结合的通信方式，任务发起者一旦宕机，同样也无法恢复。综上，这点不足也正是业界亟待解决的难题。

6.9 完成过程中遇到的困难

- 起初在 64 位 Linux 上编译 inferno 源代码无法通过，还有在移植到 arm 平台上遇到 bug，通过查阅资料，并和 inferno 官方人员交流，最终成功将其移植到 64 位的平台和 Raspberry Pi 上。
- 由于 inferno 系统普及程度不高，导致可以参考的资料非常匮乏，几乎只有官方网站上仅有的一点资料，给深入地学习带来困难，需要不断自己摸索尝试。
- inferno 系统提供的 shell 脚本语言功能有限，给代码编写造成困难，比如 inferno 的 shell 语言没有数组，导致 map 和 merge 函数并行部分结束之后必须进行表排序，降低了执行效率。
- inferno 系统偶尔出现命令执行结果不稳定的 bug，导致 3 个并行函数中作为计算完成标志的 finish 文件不能删除，然而不影响计算结果的正确性，令人比较困惑。还会出现输入运行程序命令后，一段时间没有反应，输入若干次回车之后才有反应的情况。
- 起初在三台以上节点上运行时，file2chan 分发任务不正确，经过查阅资料、阅读源代码，并做了大量实验之后，找到了解决方法，可以正确运行在多台机器上。
- 由于 inferno 系统不是很完善，代码执行出错后提供的错误信息很少，并且调试工具不完善，导致程序调试困难，只能通过检查代码和输出变量的方式，降低了开发的效率。

7 致谢

本项目是在邢凯老师的悉心指导下完成的。

首先十分感谢邢凯老师，给与我们极大的关心与帮助，为我们提供许多问题解决的方案。其次十分感谢 Inferno 原作者 Charles Forsyth，感谢他与我们进行了有益的探讨，一同解决了 Inferno 在 amd64 下编译的问题以及在 arm 环境下 Inferno 编译和网络接口问题。感谢 Pete Elmore 的博文为我们带来灵感。感谢安虹老师为我们提供研究场所，让我们专心研究课题。感谢张智帅学长为我们提供树莓派以及嵌入式开发板。感谢余家辉学长与

我们进行了众多有益的探讨。最后感谢队友之间的集思广益、通力合作，以最高效的方式解决了各种棘手的问题。

References

- [1] [https://en.wikipedia.org/wiki/Inferno_\(operating_system\)](https://en.wikipedia.org/wiki/Inferno_(operating_system)), Inferno wikipedia
- [2] https://en.wikipedia.org/wiki/Plan_9_from_Bell_Labs, Plan 9 wikipedia
- [3] http://www.vitanuova.com/inferno/info/Inferno_overview.pdf, Inferno Overview
- [4] http://www.vitanuova.com/inferno/info/Inferno_info.pdf, Inferno Information Sheet
- [5] <http://www.vitanuova.com/inferno/papers/bltj.html>, The Inferno Operating System
- [6] <http://www.vitanuova.com/inferno/papers/styx.html>, The Styx Architecture for Distributed Systems
- [7] <http://www.vitanuova.com/inferno/man/index.html>, Inferno Manual Pages
- [8] The Inferno Shell, Roger Peppé
- [9] <http://debu.gs/entries/inferno-part-0-namespaces>, Inferno Part 0: Namespaces
- [10] <http://debu.gs/entries/inferno-part-1-shell>, Inferno Part 1: Shell
- [11] The comparisons of opencl and openmp computing paradigm, SL Chu, CC Hsiao
- [12] 不同层次 MPI 并行程序容错的比较, 赵毅, 曹宗雁, 朱鹏, 迟学斌, 《e-Science 技术》2011 年 11 月第 2 卷第 6 期

- [13] Fault Tolerant MPI, Graham E. Fagg, Jack J. Dongarra, FT-MPI, Supporting Dynamic Applications in a Dynamic World
- [14] Fault-Tolerant Metacomputing and Generic Name Services: A Case Study, David Dewolfs, Jan Broeckhove, Vaidy Sunderam, Graham E. Fagg, FT-MPI,
- [15] A framework for parallel distributed computing, V. S. Sunderam, PVM
- [16] PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing, Kevin A. Huck, Allen D. Malony
- [17] Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhajan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, Timothy S. Woodall
- [18] Teaching Parallel & Distributed Computing with MPI, Joel C. Adams, Richard A. Brown, Elizabeth Shoop
- [19] A parallel sampling based clustering, Aditya AV Sastry, Kalyan Netti
- [20] GPGPU Computing Bogdan Oancea, Tudorel Andrei, Raluca Mariana Dragoescu
- [21] Research on Multi-Core PC Parallel Computation Based on OpenMP Lan Xiaowen
- [22] Design of OpenCL framework for embedded multi-core processors, Jung-Hyun Hong, Young-Ho Ahn, Byung-Jin Kim, Ki-Seok Chung