文章编号:1001 - 9081(2007)08 - 1987 - 04

P4 并行环境的建立过程分析

苗长征, 郑全录, 吴伟峰
(信息工程大学 信息工程学院, 郑州 450002)
(mcz1031@yahoo.com.cn)

摘 要:在使用 Linux 构建的工作站集群环境下,对支持并行编程的 C 函数库 P4(Portable Programs for Parallel Processors)的初始化过程进行了研究,重点剖析了进程创建和通信建立这两个部分,并对源代码中涉及到的主要数据结构及其使用给出了详尽的分析说明。对研究和开发并行编程工具有一定的参考价值。

关键词:P4;并行;消息传递界面
中图分类号: TP311.52 **文献标志码**:A

Analysis of building procedure of parallel environment with P4

MIAO Chang-zheng, ZHENG Quan-lu, WU Wei-feng
(*Institute of Information Engineering, Information Engineering University, Zhengzhou Henan* 450002, *China*)

Abstract: In the environment of workstation cluster which is configured with Linux, the initialization process of the P4 (Portable Programs for Parallel Processors) which supports the parallel programming was studied. The two parts of the creation of processes and the establishment of communication were analyzed in depth, and the use of the main data structure involved in the source code was analyzed in detail. It has a certain reference value to research and development of parallel programming tool.

Key words: P4; parallel; Message-Passing Interface (MPI)

1 背景介绍

消息传递界面(Message-Passing Interface, MPI)是一个用于并行程序开发的消息传递接口标准,得到几乎所有并行计算机厂商的支持,现已成为事实上的并行编程标准。目前国内大多文献多研究如何利用 MPI 编写并行程序,对 MPI 底层实现方法的研究较少。

P4 是用于并行程序设计的 C 函数库,是 Portable Programs for Parallel Processors 的简称。在 MPI 标准出现后,P4 被 MPICH 所吸收,成为 MPICH 在工作站集群环境下的主要实现。

P4 具有良好的兼容性,支持绝大多数版本的 UNIX 及 Linux 操作系统,不仅适用于常见的工作站集群,对 TC-2000、RS6000 等商用并行机也提供支持。

P4 支持 SPMD 和 MPMD 两种编程模式。在实现 MPMD 编程模式时,通常在各台机器上仍使用同一个程序,在程序中通过对进程标识号进行判断产生分支来实现 MPMD 的编程效果。P4 在一台机器上创建的进程数不能多于该机器所拥有的处理器的数目,但当运行环境支持 systemV IPC 时,P4 允许创建更多的进程。

用户在运行 P4 程序前,要先创建一个配置文件,P4 将根据该文件建立并行环境。配置文件中,每行表示一个节点,包含四项内容:节点所在主机名、指定要创建的进程数、可执行程序的路径和运行程序的用户名。其中,节点和主机是两个不同的概念。如果一个主机在配置文件中重复出现多次,将被认为是多个不同的节点。

由于 P4 支持多种操作系统和多种体系结构的机器,所以对于不同的实验环境和参数设置,并行环境建立过程并不完全相同。因此,分析将以下面描述的具体实例为基础而展开。

使用的实例为:3 台单处理器 PC 机,主机名分别为 node1,node2,node3,操作系统为 Linux(内核版本 2.6),登录用户名为 user,支持 systemV IPC,主机间用以太网连接,各机器上存储有相同的可执行程序,程序名为 a.out,存放路径为/home/user/。用户要求创建 9 个进程,每个节点 3 个。在 node1 上执行 a.out,并编辑配置文件内容如下:

```
node1  2  /home/user/a.out  user
node2  3  /home//user/a.out  user
node3  3  /home/user/a.out  user
```

用户直接控制的节点称为主节点,即 node1。对于主节点,在配置文件中指定的进程数目要比实际数目少 1,因此配置文件中 node1 的第二项是 2 而不是 3。

2 并行环境建立过程

2.1 创建进程

2.1.1 进程的分类

P4 在初始化过程中创建的进程可分为用户进程和监听进程。用户进程由用户在配置文件中显式指定。监听进程是 P4 为实现通信而隐式创建的,对用户是不可见的,每个节点只有一个监听进程。P4 用 struct local_data 中的整型变量 my_id 对这两种进程进行区分。对于用户进程,my_id 取值为大于等于 0 的整数,对于监听进程,my_id 取值为 -99。

对于用户进程,又可分为主进程和从过程。节点上创建

收稿日期:2007 - 03 - 01;**修回日期**:2007 - 05 - 05。

作者简介:苗长征(1978 -),男,河南新乡人,硕士研究生,主要研究方向:并行计算; 郑全录(1965 -),男,河南安阳人,副教授,博士,主要研究方向:并行计算; 吴伟峰(1981 -),男,河南郑州人,硕士研究生,主要研究方向:软件技术、逆向工程。

的第一个进程为主进程,其余为从进程。P4 用 struct proc_info 中的布尔变量 am_rm 对这两种进程进行区分。对于主进程,该变量取值为真,从进程反之。

主进程又可分为主节点主进程和远端节点主进程两种。由用户手工创建的进程就是主节点主进程,也称为 big master。除此之外,其他的主进程均为远端节点主进程,也称为 remote master。远端节点主进程由主节点主进程负责创建,主节点主进程由用户手工直接创建。P4 用 struct local_data 中的布尔变量 am_bm 对这两种进程进行区分,对于主节点主进程,该变量取值为真,远端节点主进程反之。

就本文实例来说,进程创建的过程是这样的:用户在 node1 上运行 a.out,生成 0 号进程,0 号进程在 node1 上创建 1 号、2 号进程和监听进程,在 node2 上创建 3 号进程,在 node3 上创建 6 号进程。3 号进程在 node2 上创建 4 号、5 号进程和监听进程。6 号进程在 node3 上创建 7 号、8 号进程和监听进程。其中,0 号进程为主节点主进程,3 号、6 号进程为远端节点主进程,1 号、2 号、4 号、5 号、7 号、8 号进程为从进程。在 node1、node2、node3 上各有一个监听进程。下面对各进程的具体创建过程进行分析。

2.1.2 主节点主进程(0 号)

0 号进程由用户手工创建,负责本地节点监听进程、本地节点从进程(1 号、2 号)和远端节点主进程的创建(3 号、6 号)。其所作的主要工作如下:

1)处理命令行参数,为 local_data、p4_global_data 等数据结构分配空间并初始化,将自身信息填入 p4_global -> proctable[0]。

2)读取配置文件,并将读入的内容写入 p4_local -> procgroup。

3)根据 p4_local -> procgroup 的内容细化 p4_global -> proctable[]。

4)创建主动套接字,设为 socket1。

5)创建管道,设为 pipe1,其两个端口分别为 end1 和 end2(下同)。

6)调用 fork(),创建 1 号进程。

7)调用 close(),关闭 pipe1 的 end1 端口。

8)将 1 号进程信息写入 p4_global -> proctable[1]。

9)重复(5)至(8),创建 2 号进程和 pipe2,并将其信息写入 p4_global -> proctable[2]。

10)创建管道,设为 pipe3。

11)调用 fork(),创建监听进程。

12)调用 close(),关闭 socket1。

13)调用 close(),关闭 pipe3 的 end2 端口。

14)创建主动套接字 socket2。

15)调用 fork() 和 execlp(),通过 rsh 或 ssh 命令运行 node2 上的 a.out 文件,在 node2 上创建 3 号进程。node1 的主机名和 socket2 的端口号均通过命令行参数传递给 3 号进程。

16)调用 accept(),在 socket2 上接受来自 3 号进程的连接请求,与 3 号进程建立 TCP 连接,设为 socket3。

17)通过 socket3,将由配置文件读入的有关的初始化信息发送给 3 号进程。

18)在 socket3 上接收由 3 号进程发送的 node2 上各进程(3 号、4 号、5 号以及监听进程)的信息,并填写 p4_global -> proctable[3] ~ p4_global -> proctable[5]。

19)重复 15) ~ 18),创建 6 号进程,与 6 号进程建立 TCP 连接,并填写 p4_global -> proctable[6] ~ p4_global -> proctable[8]。至此,p4_global -> proctable[]内容填写完毕。

20)调用 close(),关闭 socket2。

21)通过 socket3,将 p4_global -> proctable[]内容发送给 3 号进程。同理,通过与 6 号进程之间的 TCP 连接将该内容也发送给 6 号进程。

22)与本地节点从进程和远端节点主进程进行同步操作。

2.1.3 远端节点主进程(以 3 号为例)

1)处理命令行参数,获取 node1 上 socket2 的相关信息。

2)调用 connect(),通过 socket2 与 0 号进程建立 TCP 连接,设为 socket4。

3)在 socket4 上接收 0 号进程发送的初始化信息。

4)为 local_data、p4_global_data 等数据结构分配空间并初始化。

5)创建主动套接字,设为 socket5。

6)通过 socket4,将 socket5 以及 3 号进程自身信息发送给 0 号进程。

7)创建管道,设为 pipe4。

8)调用 fork(),创建 4 号进程。

9)调用 close(),关闭 pipe4 的 end1 端口。

10)通过 socket4,将 4 号进程信息发送给 0 号进程。

11)重复(7)至(10),创建 5 号进程和管道 pipe5,并将 5 号进程的信息发送给 0 号进程。

12)创建管道,设为 pipe6。

13)调用 fork(),创建监听进程。

14)调用 close(),关闭 socket5。

15)调用 close(),关闭 pipe6 的 end2 端口。

16)通过 socket4,接收 0 号进程发来的 p4_global -> proctable[]的内容。

17)与本地节点从进程和 0 号进程进行同步操作。

2.1.4 从进程(以 1 号为例)

1)为 local_data 数据结构分配空间并初始化。

2)调用 close(),关闭 pipe1 的 end2 端口。

3)调用 close(),关闭 socket1。

4)与 0 号进程进行同步操作。

2.1.5 监听进程(以 node1 节点为例)

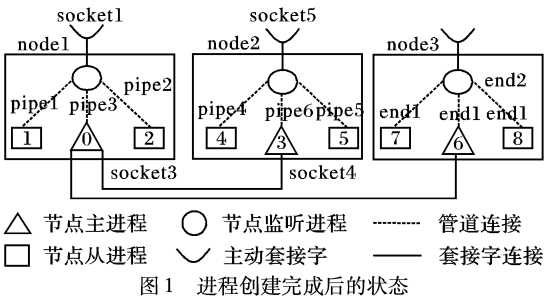

图 1 进程创建完成后的状态

1)调用 close(),关闭 pipe3 的 end1 一端。

2)初始化 local_data 结构。

3)进入 while 循环,通过调用 select()对来自 socket1 和 pipe1、pipe2 和 pipe3 的连接请求进行处理。

至此,P4 进程创建工作完成,进程分布及进程间通信连接建立情况如图 1 所示。

2.2 建立通信

2.2.1 通信的种类

进程间通信可分为节点内部和节点间两种。节点内部通信通过共享内存方式实现。节点间通信通过流套接字即 TCP 协议实现。

按照接收方和发送方进程数量的不同,进程间通信又可分为一对一、一对多、多对一和多对多四种。由于流套接字不支持广播操作,因此 P4 中的群集通信都是以一对一通信为基础实现的。所以,下面仅讨论一对一收发操作这种情况。

2.2.2 节点内部通信

进程在节点内部的通信通过共享内存方式完成。以 1 号进程向 2 号进程发送消息为例。

1)1 号进程在共享内存中申请空间创建 struct p4_msg 结构,将消息发送的源、目的地址等控制信息以及消息正文放入其中。

2)1 号进程将上面创建的 p4_msg 加入到 p4_global -> shmem_msg_queues[2]所描述的消息队列链表中。

3)2 号进程根据消息源进程的标识号,从 p4_global -> shmem_msg_queues[2]所描述的 p4_msg 链表中查找到指定的 p4_msg 结构体,从中取出消息,完成接收操作。

2.2.3 节点间通信

进程在节点间的通信通过流套接字实现。

对于已建立好流套接字连接的进程,如 0 号和 3 号进程之间,P4 就直接调用套接字 API 完成消息收发。如果连接尚未建立,如 2 号进程和 5 号进程之间,P4 将先建立连接,然后再完成收发(也可以通过参数设置使 P4 在进程创建过程中就完成所有通信连接的建立)。

下面就以 2 号进程向 5 号进程发送消息为例对 P4 建立流套接字连接的过程进行分析。

1)2 号进程在执行消息发送语句时,检测发现和 5 号进程之间的连接尚未建立。

2)2 号进程根据 p4_global -> proctable[5]找到 5 号进程所在节点 node2 上的主动套接字 socket5 的端口号。

3)2 号进程通过 socket5 与 node2 上的监听进程建立连接,设为 socket6。

4)2 号进程创建一个主动套接字 socket7。

5)2 号进程通过 socket6 将 socket7 的端口号发送给 node2 上的监听进程。

6)node2 上的监听进程接收到 2 号进程发送来的 socket7 的端口号后,通过 pipe5 将其发送给 5 号进程。

7)5 号进程创建被动套接字 socket8,通过 socket7 与 2 号进程建立连接。

8)2 号进程接受 5 号进程的连接请求,在 socket9 上与 5 号进程的 socket8 建立连接。

9)2 号进程调用 close(),关闭 socket7。

10)通信连接建立完成,2 号进程通过 socket9,5 号进程通过 socket8 进行消息收发。

在通信建立过程中,有两点需要指出:一是 P4 在节点间建立连接时要求必须按照上述方式进行,即由 id 号小的进程建立主动套接字。在上例中,如果是 5 号进程向 2 号进程发出连接请求的话,由于 5 大于 2,所以 P4 会将其转变为 2 号进程向 5 号进程发出请求的模式之后再进行处理。二是在发送消息时,为避免死锁发生,P4 会首先检查源进程 id 号和目的进程 id 号的大小。如果源进程 id 号大于目的进程 id 号,源进程在执行发送操作之前,要检测是否有消息到达。如有消息到达,就执行接收操作,接收该消息并存储到自己的消息队列中,然后再执行发送操作;如果没有消息到达,则直接执行发送操作。该规则在一定程度上预防了死锁的发生。例如 2 号进程和 4 号进程之间互发消息,而且在程序中两个进程都是按照先发后收的顺序执行,这时如果消息长度足够大(一次发送操作无法全部写入内核缓冲区),则很容易陷入死锁状态。采用上述规则后,2 号进程仍将是先发送后接收,但 4 号进程在执行发送语句时所做的动作实际上是先接收再发送,从而避免了死锁。4 号进程在接下来的接收语句中并没有执行真正的接收操作,只是在本地消息队列中调出刚才已接收过的消息。

3 并行环境建立中用到的主要数据结构

在 P4 建立并行环境的过程中,主要用到了以下数据结构。

3.1 struct local_data * p4_local

local_data:用于存储每个进程所独有的信息,其主要成员有:

int my_id:用于存储该进程在全局进程组中的编号,是该进程与其他进程相区别的唯一标识,由 P4 在初始化过程中按照一定规则分配,其取值范围为一正整数,范围从 0 到用户进程总数减去 1。对于节点监听进程,该变量取值均为 -99。

struct p4_procgroup * procgroup:p4_procgroup 结构体用于存储 0 号进程读入的配置文件的内容。对于 my_id 不等于 0 的进程,由于不承担读取配置文件的任务,因此 procgroup 指针均被置为 NULL。

```
struct p4_procgroup
{ struct p4_procgroup_entry entries[N];      // N 代指常数,下同
  int num_entries;                           // 配置文件中节点数量
};
struct p4_procgroup_entry                    // 具体节点信息
{ int numslaves_in_group;                    // 创建的进程数
  int rm_rank;                               // 节点上主进程的全局编号
  char host_name[N];                         // 节点所在主机名
  char slave_full_pathname[N];
                                             //节点上可执行程序名及绝对路径
  char username[N];                          // 运行程序的用户名
};
```

int listener_fd:进程与所在节点监听进程通过管道建立连接后,在进程一端的管道描述符就存储在该变量内。

struct connection * conntab:该指针指向的是一个 connection 类型的数组,用于存储进程与其他进程连接的基本信息,其定义如下:

```
struct connection
{ int type;                                  // 连接类型
  int port;                                  // 连接描述符
  ...
}
```

type 表示与对应进程连接的类型,对于同一节点上的进程,为 CONN_LOCAL,对于不同节点上的进程,如果连接已经建立,为 CONN_REMOTE_EST,否则为 CONN_REMOTE_NON_EST。如果连接已经建立,则 port 为该连接对应的描述符,否则 port 无意义。

bool am_bm:该逻辑变量用于指明该进程是否是 0 号进程,bm 意为 big master,即由用户直接启动的进程。

struct p4_msg_queue * queued_messages:该指针指向一个消息队列,进程接收的消息首先存放在该队列中,尔后再进行处理。

3.2 struct p4_global_data * p4_global

p4_global_data 由节点主进程利用 systemV IPC 在共享内存中申请空间创建,对节点上所有进程可见。其主要成员有:

struct proc_info proctable[N]:该数组负责存储全局所有进程的详细信息。在初始化过程中由 0 号进程负责填写,并发送其副本给其他节点。

```
struct proc_info                    // 存储各进程的详细信息
{ int unix_id;                      // 进程的 PID
  int slave_idx;                    // 进程在节点中的编号
  int group_id;                     // 进程所在节点的编号
  bool am_rm;                       // 是否是节点上的主进程
  char host_name[N];                // 所在节点主机名
  …
}
```

int listener_pid 和 int listener_fd:存储节点监听进程的 PID 和由监听进程管理的主动套接字的描述符。

struct p4_msg_queue shmem_msg_queues[N]:该数组为节点上的每个进程创建一个消息队列结构,用于节点内部进程间收发消息。

int num_in_proctable:该变量存储全局进程的总数目,但节点监听进程不包括在内。

3.3 struct listener_data * listener_info

listener_data:由节点主进程创建,节点监听进程通过 fork()从主进程处继承并负责管理。其主要数据成员有:

int listening_fd:存储由监听进程管理的主动套接字的描述符。

int num:存储该节点上由用户管理的进程数目。

int * slave_fd:该指针指向一个整型数组,其长度由 num 变量确定。该数组用于存储监听进程和节点上用户进程相连的管道在监听进程一端的描述符。

int * slave_pid:该指针指向一个整型数组,其长度由变量 num 确定。该数组用于存储节点上用户进程的 PID。

4 结语

P4 根据机器体系结构和参数设置的不同,其并行环境建立过程也不尽相同。以上只是对其中较为常见的情况进行了分析,接下来将在现有研究的基础上尝试做一些有意义的改进和创新。

参考文献:

[1] 陈国良. 并行算法实践[M]. 北京:高等教育出版社,2004.

[2] 蒋砚军,高占春. 实用 UNIX 教程[M]. 北京:清华大学出版社,2005.

[3] GRAY J S. UNIX 进程间通信[M]. 2 版. 张宁,译. 北京:电子工业出版社,2001.

[4] MOLAY B. UNIX/Linux 编程实践教程[M]. 杨宗源,黄海涛,译. 北京:清华大学出版社,2004.

[5] COMER D E, STEVENS D L. 用 TCP/IP 进行网际互联第三卷:客户—服务器编程与应用(Linux/POSIX 套接字版)[M]. 赵刚,林瑶,蒋慧,等译. 北京:电子工业出版社,2001.

(上接第 1986 页)

模块化 例外处理不需要对基础流程进行修改,基础流程设计人员可专注于流程本身的设计与优化,保持基础流程的简单和清晰;

可扩展性 新的例外情况出现时,可以通过增加新的规则、新的基础流程以及新的元流程进行处理,从而使例外处理具有良好的可扩展性,该特性尤其适合动态 Web 服务组合的例外处理;

适应性 通过对规则进行适当的增加、删除和修改,可以使服务组合流程以一种简单的形式适应不可预见例外等特定的情况。

此次提出的框架中约定实施满足条件的第一条规则,从而避免了规则的冲突。因此,对更复杂的规则冲突的解决是下一步研究的重点。

参考文献:

[1] 岳昆, 王晓玲, 周傲英. Web 服务核心支撑技术:研究综述[J]. 软件学报, 2004, 15(3):428 - 442.

[2] TSALGATIDOU A, PILIOURA T. An overview of standards and related technology in Web services[J]. Distributed and Parallel Databases, 2002, 12(2):135 - 162.

[3] 饶元, 冯博琴, 李尊朝. 基于 Web Services 的服务合成技术研究综述[J]. 系统工程与电子技术, 2005, 27(8):1481 - 1489.

[4] 刘必欣. 动态 Web 服务组合关键技术研究[D]. 长沙:国防科学技术大学, 2005.

[5] MILANOVIC N, MALEK M. Current solutions for Web service composition[J]. IEEE Internet Computing, 2004, 8(6):51 - 59.

[6] KORHONEN J, PAJUNEN L, PUUSTJÄRVI J. Automatic composition of Web service workflows using a semantic Agent[C]// Proceedings of the IEEE/WIC International Conference on Web Intelligence. Washington: IEEE Computer Society, 2003: 566 - 569.

[7] Van Der AALST W M, DUMAS M. Web service composition languages: old wine in new bottles?[C]// Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture". Washington: IEEE Computer Society, 2003: 298 - 307.

[8] EDER J, LIEBHART W. The workflow activity model WAMO [C]// Proceedings of the 3rd International Conference on Cooperative Information Systems. Toronto: University of Toronto Press, 1995: 87 - 98.

[9] HAGEN C, ALONSO G. Exception handling in workflow management systems[J]. IEEE Transaction on Software Engineering, 2000, 26(10):943 - 958.

[10] SHI Y L, ZHANG L, SHI B L. Exception handling of workflow for web services[C]// Proceedings of the Fourth International Conference on Computer and Information Technology. Washington: IEEE Computer Society, 2004: 273 - 277.

[11] Van Der AALST W M, Van HEE K. 工作流管理—模型、方法和系统[M]. 北京:清华大学出版社, 2004.

[12] CAO J N, YANG J, CHAN W T. Exception handling in distributed workflow systems using mobile Agents[C]// Proceedings of the IEEE International Conference on e-Business Engineering. Washington: IEEE Computer Society, 2005: 48 - 55.

[13] ANDREWS T, CURBERA F, DHOLAKIA H. Business process execution language for web services version 1.1[EB/OL]. http://www-106. ibm. com/developerworks/webservices/library/ws-bpel/, 2003.