

一种异构并行计算仿真模型设计

刘清涛, 于立新

(北京微电子技术研究所 SoC 部, 北京 100076)

摘 要: 为了加快异构体系调度算法的验证速度, 降低时间成本和验证复杂度, 本设计采用软件仿真技术建立了一种异构体系多任务并行执行模型. 针对调度算法运行的硬件环境不同, 使用 ERT 矩阵、通信因子矩阵、输出量数组和机器时间数组抽象出了硬件并行执行模型. 通过建立存储器及通信端口模型, 用串行语言模拟了异构体系中任务的并行执行过程, 实现了通信数据的转送和乱序调度, 从而加快了调度算法的验证速度. 为增加可移植性, 模型设计时未使用任何操作系统的系统调用.

关键词: 异构体系; 任务调度; 软件仿真; 并行模型

中图分类号: TP319

文献标识码: A

文章编号: 1000-7180(2013)11-0143-06

A Parallel Execution Model of Heterogeneous Architecture

LIU Qing-tao, YU Li-xin

(Beijing Microelectronics Technology Institute SoC Department, Beijing 100076, China)

Abstract: To accelerate verification speed of scheduling algorithm in heterogeneous architecture and reduce time consumption and verification complexity, this design utilizes software simulation technique to build a multi-task parallel execution model for heterogeneous architecture. This technique uses ERT matrix, communication factor matrix, output array and machine time array to simulate different hardware environment. A parallel model is implemented via serious language which has the function of data forwarding and out-of-order scheduling. No system API is referred considering portability in the designing process.

Key words: heterogeneous architecture; task scheduling; software simulation; parallel model

1 引言

作为并行处理的一个主要分支, 异构计算体系在科研、商业、工业方面起到了不可替代的作用. 要充分发挥异构体系的性能, 需要有效的调度算法. 当前针对异构体系已经提出了很多的调度算法. 但为给定硬件选择一个合适的调度算法依然很困难, 因为不同的算法通常基于不同的假设. 因而设计一个高效的调度算法验证模型来进行算法验证或进行算法对比就变得十分必要.

异构体系中, 不同机器间交换信息会产生通信延迟. 随着机器计算能力的提升, 通信延迟逐渐成为并行处理中的速度瓶颈. 因而本文在建立异构体系并行模型时考虑了通信延迟.

2 任务调度相关定义和假设

为了建立仿真模型, 需要做出一些约束和假设^[1-3], 如果真实的硬件环境与本模型所模拟的环境有所差别, 可通过设置相关参数或者更改模型的对应功能模块来得到与新硬件相匹配的模型. 本文使用了下面所述的一些假设^[4-5].

假设任务间的依赖关系已知, 依赖关系用有向无环图(DAG)表示, 如图 1 所示. 定义 $|S|$ 为任务群中任务的数量. s_i 为第 i 个任务. 定义 $|M|$ 为机器数量, m_j 为第 j 台机器(或多核处理器中的第 j 个处理器核). 假设各任务在每台机器上的运行时间已知, 这些时间存储在一个大小为 $|S| \times |M|$ 的矩阵 ERT (estimated run time) 中. 元素 $ERT(i, j)$ 表示任务 i

在机器 j 上的估计运行时间. 文献 [2,5] 中描述了估计任务运行时间的方法.

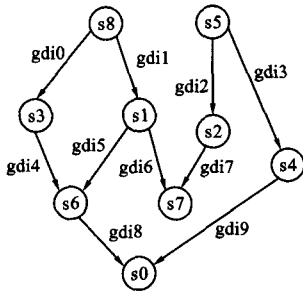


图 1 有向无环图 (DAG)

本文用 gdi(global data item)表示通信延迟. 不同的通信路径对应不同的 gdi. 任务的输出量用一个长度为 $|S|$ 的向量 $Output$ 来表示. $Output(i)$ 代表任务 i 的输出. 任务没有输出时, $Output$ 对应元素值为 0. 任务间依赖关系用 $|S| \times |S|$ 的邻接矩阵^[6] Adj 表示. $Adj(i, j)$ 为 0 表示任务 i 和 j 之间无依赖关系; $Adj(i, j)$ 为 1 表示任务 j 依赖于任务 i . 本文还假设不同机器间任务的执行及数据的传输是非抢占的, 这一假设在多数异构体系中都成立.

虽然一个任务的输出量一定, 但当这一输出量传输到不同机器时产生的延迟是不同的. 本文用一个 $|M| \times |M|$ 的通信因子矩阵 $Comm$ 来描述这一特性. $Comm(m, n)$ 表示机器 m 到机器 n 通信路径的延迟因子. 假定在机器 m 上的任务 i 要向在机器 n 上的任务传输数据, 那么其产生的延迟为:

$$gdi = Output(i) * Comm(m, n)$$

3 调度的表示

为了表示清晰, 将一个调度分为两个独立的部分: 匹配串和调度串. 匹配串表示任务到机器的分派, 调度串规定了任务的调度顺序. 图 2 中显示了图 1 对应两个合法的调度结果.

4 执行模型的构建

本文使用 C 语言构建异构环境的并行执行模型, 且使用文献[7]中的算法描述方法.

4.1 硬件模型

硬件模型主要由以下三种数据结构来实现:

(1) 用数组 $intmachTime[MACH_NUM]$ 来记录机器时间, $MACH_NUM$ 表示处理机(处理器)数量. 等待 gdi 传送或有任务执行时, 机器时间前进.

(2) 用链表数组 $t_listinPortIdle[MACH_NUM]$ 表示各机器的输入端口, 其中存放输入端口

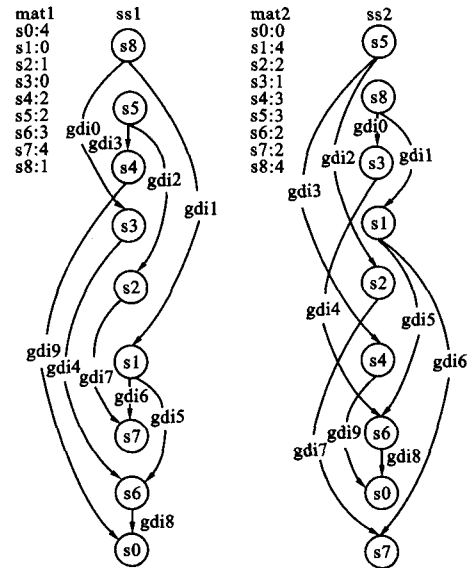


图 2 图 1 对应的两个合法调度

的空闲时间窗口. 链表每个节点中是一个指向如下结构体的指针:

```
typedef struct
{
    int startIdle;
    int endIdle;
} T_PortNode;
```

$startIdle$ 表示空闲时间窗口的起始时间, 成员 $endIdle$ 表示空闲时间窗口的结束时间. 一个链表中时间窗口的时间依次递增

(3) 用链表数组 $t_listOutPortIdle[MACH_NUM]$ 表示各机器的输出端口, 其中存放输出端口的空闲时间窗口. 链表每个节点中是一个指向上面结构体的指针. 链表中包含的时间窗口的时间是依次递增的.

两个端口数组能模拟通信端口的忙闲状态, 从而能对可用的通信窗口进行搜索和选择, 实现全局通信数据的乱序调度. 在端口数组中, 每个链表最后一个节点的 $endIdle$ 是 C 语言中 int 类型能表示的正整数的上限, 其作为一个哨兵来表示机器端口的最后一个空闲时间窗口的上限是正无穷大.

机器时间数组能模拟对 gdi 的等待和任务在处理器(处理器)上的执行. 如果一个 gdi 的传送完成时间比当前目标时间机器大, 那么就将目标机器时间更新为此 gdi 传输完成的时间, 表示目标机器的任务在等待所需要的 gdi 传输完成. 当依赖的所有 gdi 都输送完成之后, 才能开始执行这个任务. 在任务所在机器的机器时间上加上任务的执行时间表示任务的执行完成.

4.2 数据转送

异构体系中除了全局存储器,通常还包含机器的局部存储器^[8]. 局部存储器会存储接收到的数据并当其他任务需要相同数据时把数据转送出去. 如果某台机器没有足够的局部存储器,可以给本模型中的转送链表设置一上限,表示局部存储器的大小. 达到上限时,根据硬件实际情况关闭数据转送或对数据进行替换. 本文假设所有的机器都有足够的局部存储器,以保证在局部存储器不足时进行限定和调整. 当一个任务需要某个数据时,将从此数据的初始源任务及能转送此数据的任务中选择能最早完成传输的作为实际的源任务. 数据转送能减少整个任务群的执行完成时间^[4].

用链表数组 `t_list forward[TASK_NUM]` 实现数据转送功能,转送数组索引对应任务号. 若任务无输出,对应的数组元素为空指针 `NULL`. 否则 `forward[i]` 表示的链表中存放到目前为止,能转送任务 `i` 输出量的所有任务. 链表节点指向的结构体如下:

```
typedef struct
{
    int taskNum;
    int avlTime;
} T_PortNode;
```

其中, `taskNum` 表示转送任务的任务号, `avlTime` 表示 `gdi` 的传输完成时间,即 `gdi` 在此任务所在机器上的开始可用时间. 每当完成一个全局数据项传送,就在初始源任务(其输出被转送的任务)对应的转送链表中加入一新节点,新节点的 `taskNum` 为 `gdi` 传输的目标任务,新节点的 `avlTime` 为数据开始传输时间与数据传输花费的时间之和. 下面给出实现数据转送的算法:

A1 [查找是否在同一机器上] 查找初始源任务和初始源任务对应转送链表中的转送任务,如果初始源任务或某个转送任务与目标机器在同一机器上(查看匹配串),使用这个传输,返回源机器号,算法结束. 否则转到 A2.

A2 [计算初始源任务传输完成时间] 计算使用初始源任务的传输完成时间(调用算法 4.4B). 如果完成时间小于目标机器时间,使用这个传输,转到 A5,否则转到 A3.

A3 [使用转送任务传输] 依次计算各转送任务进行传输时的完成时间. 如果某个转送任务完成传输的时间(考虑了乱序调度,见算法 4.4B)在目标机

器之前,就使用这个传输,转到 A5. 否则转到 A4.

A4. [使用最早可用窗口] (此时不论初始源任务还是转送任务的传输完成时间都比目标机器时间大,目标机器要等待传输完成)对比初始源任务和各转送任务的传输完成时间,以其中最早完成传输的任务作为此次传输的源任务.

A5. [更新转送链表] 将目标任务对应的结构体加入初始源任务的转送链表. 结构体 `taskNum` 成员为目标任务的任务号,结构体 `avlTime` 成员为传输的完成时间,算法结束.

步骤 A1 中目标任务和源任务(初始源任务或转送任务)在同一机器上时,完成传输却没有将目标任务加入到初始源任务转送链表中的原因是源任务已经能对相同数据进行转送,且数据可用时间较目标任务更早.

4.3 乱序调度

全局数据的乱序调度^[4]能减少任务群的整体执行完成时间. 全局数据项的乱序调度分为最早可用窗口乱序调度和最小可用窗口乱序调度. 假设有 2 台机器 `m0` 和 `m1`,其中的任务调度分派如图 3(a)所示. 对 `gid2` 使用乱序调度且选择最早可用窗口的通信窗口,其调度结果如图 3(b)所示,对 `gdi2` 选择最小可用窗口的结果如图 3(c)所示. 对比图 3(b)和图 3(c)可以发现,图 3(c)前 10 秒中 `m0` 的输出端口出现了一个长度为 3(3 到 6 秒)的空闲时间段,而图 3(b)中的 `m0` 的输出端口在前 10 秒内却没有大小为 3 的端口空闲时间. 从统计学角度看,最早可用窗口调度时间相对于最小可用窗口调度会产生更多的小窗口,这会减小输出端口和输入端口的窗口交叠区间长度,不利于端口空闲时间段的集中使用. 因而,总体而言,最小可用窗口乱序调度会有更好的调度结果,本文选择使用最小可用窗口乱序调度. 乱序调度会增加调度算法本身的运行时间(编译时间),但会减小任务群的整体执行完成时间.

4.4 单个 gdi 最小可用窗口乱序调度的实现

单个 `gdi` 的最小可用窗口乱序调度算法(只是查找最合适传输窗口,并未进行真正的传输):

B1 [查找可用窗口] 查找源机器的输出端口链表和目标机器的输入端口链表,如果在 `gdi` 可用时间之后和目标机器时间之前有两个端口中有同时可用的通信窗口,那么找到其中最小的,作为最终选择的可用通信窗口,转到 B3. 否则(在 `gdi` 可用时间之后及目标机器时间之前无可用窗口),转到 B2.

B2 [使用首组窗口] 使用查找到的首个可用窗

口进行数据传输。

B3[返回] 返回当前进行传输的两个窗口指针、窗口的交叠类型、源任务所在机器及 gdi 传输完成时间。

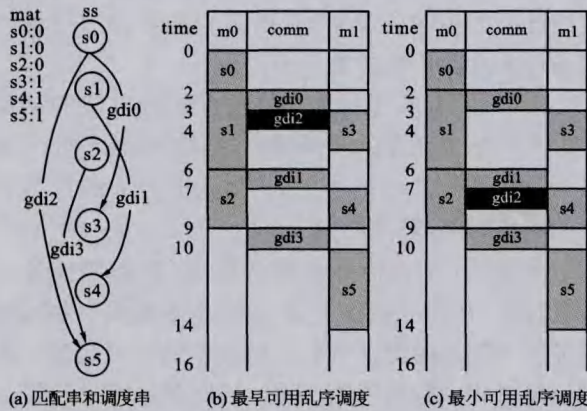


图 3 最早可用窗口与最小可用窗口乱序调度

4.5 更新通信端口链表

根据算法 4.4B 的返回结果,更新两个端口的空闲时间链表(进行真正的数据传输).更新端口链表时,不同的窗口交叠种类要对端口链表进行不同的处理.图 4 和图 5 是可用窗口与 avlTime 不相交时,9 类窗口交叠情况.例如,针对图 4(a)的情形,更新输出端口和输入端口链表时,应将两个窗口的顶部都加上 gdi 的传输时间.对于图 4(d)的情况,更新端口链表时,应更新左窗口顶部和右窗口底部,同时在输入端口空闲时间链表中、右窗口之后添加一新的窗口,这个窗口对应 gdi 传输后将原右窗口新分割出来的窗口.如果两个窗口中较小的窗口恰好与 gdi 传输时间相同(例如图 4(e)的右窗口),那么在更新这个窗口所在的链表时,要将这个窗口对应的节点删除,表明此窗口被完全使用,无剩余空闲时间.

处理窗口交叠时,除了要考虑窗口的交叠种类,还要考虑被选择进行数据传输的一组窗口是否是离 avlTime 最近的可用窗口.avlTime 可能与窗口相交、与窗口顶部相切(底部相切时,此窗口肯定不可用)、与窗口顶部相离三种情况.由于相切和相离时对链表进行相同的处理,可以不做区分.因而,根据相交和相离(相切),avlTime 与窗口的关系可以分为图 6 所示的 4 种情形,在更新链表时,这 4 种情况要与图 4 和图 5 的 9 种情形综合考虑.

4.6 计算单个任务的完成时间

执行一个任务前要完成所有依赖 gdi 的传输.涉及的依赖链表节点指向的结构体为:

typedef struct

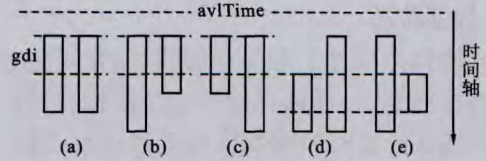


图 4 窗口交叠种类

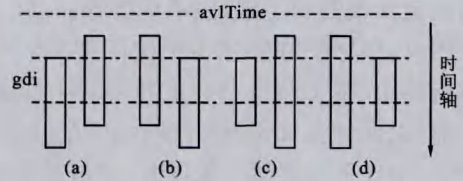


图 5 窗口交叠种类

图 6 avlTime 与首组可用窗口的相交种类

```
{
    intdpndTaskNum;
}
```

dpndTaskNum 表示被依赖的任务号.计算单个任务执行完成时间的算法如下:

C1[建立依赖链表] 扫描邻接矩阵 Adj 中以本任务为列标的列,如果此列中所有元素都为 0(表明当前任务无依赖),转到 C5.否则,建立一个依赖任务链表.当某元素的值为 1 时,新建一个依赖任务节点,将此元素的行标赋值给 dpndTaskNum,并将此节点加入依赖链表.新建一变量 intleastGdi,表示所有 gdi 中能最早完成的时间.

C2[查找最早完成的 gdi] 将 leastGdi 值赋为 int 类型能表示的最大正整数值 INTN_MAX.扫描依赖链表,若依赖链表为空(所有需要的 gdi 都传送完成),转到 C5,否则,对链表中的每个节点代表的任务,以此任务为初始源任务,当前的任务为目标任务执行算法 4.3A(得到所有依赖 gdi 的传送完成时间及对应的传输窗口和源机器),如果算法 4.3A 的完成时间在目标机器时间之前或者算法 4.3A 得到的源机器与目标机器相同,那么调度这个 gdi,转到 C4.否则(所有 gdi 的传送完成时间都在目标机器时间之后,且所有的依赖源任务及依赖任务的转送任务都与当前目标任务不在同一机器上),转到 C3.

C3[查找最早完成的 gdi] 计算所有依赖 gdi 的传送完成时间,找到能最早的 gdi,将其传输完成时间赋值给 leastGdi,调度此 gdi.

C4[更新端口链表和依赖链表] 更新当前调度

的 gdi 对应的输出端口和输入端口空闲时间链表,在依赖链表中删除当前 gdi 的源任务号对应的节点(此 gdi 调度完成),转到 C2。

C5[执行任务] 将目标机器的机器时间加上当前任务在相应机器上的执行时间(依据匹配串和 ERT 矩阵),算法结束。

4.7 计算所有任务的完成时间

对调度串中的任务按照调度顺序依次执行算法 4.6C,直到最后一个任务。查找所有机器时间中的最大值,即整个任务群的完成时间。

5 模型运行结果分析

5.1 初始化硬件及数据准备

图 7 给出了任务执行前,任务匹配、任务调度顺序、各任务的输出、通信因子矩阵和端口空闲时间窗口的情况。在图 7(b)中,各任务的输出量是一个随机数,为了便于理解,将其设置为 1 到 5 之间,如果没有输出,则设置为 0。通信因子矩阵的元素值也是随机生成的,其范围在 1 到 10 之间,本机间数据传输的时间代价为 0,因而通信因子矩阵中的对角元素都为 0。在端口列表图 7(c)中,冒号左侧的数字表示机器号,冒号右侧圆括号表示端口空闲时间窗口,一个圆括号代表一个空闲时间窗口。数字 2147483647 是 C 语言中 int 类型能表示的最大正整数,它代表时间的无穷大值,即在任务开始执行前,端口从 0 到无穷大的时间内都是空闲的。

5.2 结果分析

图 8 显示了图 7(a)表示的调度在初始化后的硬件模型上的运行结果(为方便分析,仅列出的端口情况,未显示各任务执行完成时间)。在图 8 中,当表示 gdi 传输时,标题冒号左边是要传输的 gdi 序号,冒号右边是 gdi 开始传输时,目标机器的时间。图中各端口时间是标题中的 gdi 传输完成后,各端口的可用时间窗口。

从图 8 中可以看出,任务 8 和任务 5 执行完成后,端口可用时间窗口没有变化,这是因为任务 8 和任务 5 没有依赖,这二者的执行不需要使用端口传输数据。gdi3 传输完成后,端口可用时间窗口也无变化,这是因为在匹配串中,任务 4 和任务 5 被分配到了同一机器上。任务 4 的依赖数据 gdi3 来自于 gdi5,从本机传输,不需要使用端口。

对于 gdi0,它来自于任务 8,由任务 3 接收。而任务 8 和任务 3 分别被分派到机器 1 和机器 0 上,因而 gdi3 的传输时间是:

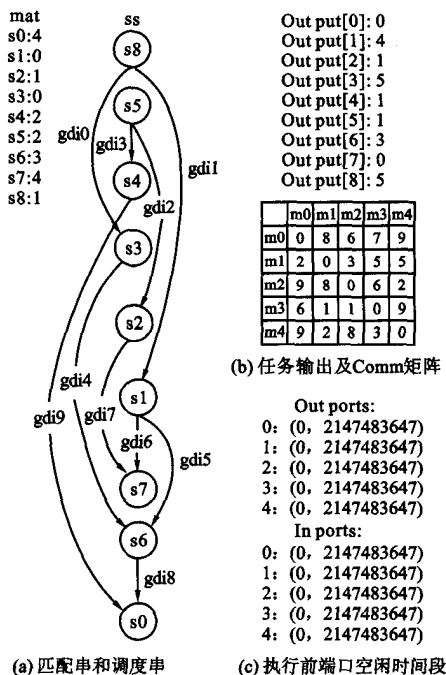


图 7 任务执行前数据准备

$$gdi0 = \text{Output}[8] * \text{Comm}[1][0] = 10$$

对比 gdi0 传输前后的输出和输入端口,可以看出,机器 1 输出端口和机器 0 输入端口的 7 到 17 秒的时间窗口被占用,这个窗口大小正好是 10 秒,与之前的计算结果一致。

对于 gdi2,它来自于任务 5,由任务 2 接收。而任务 5 和任务 2 被分别分派到机器 2 和机器 1 上,因而 gdi0 的传输时间是:

$$gdi2 = \text{Output}[5] * \text{Comm}[2][1] = 8$$

对比 gdi2 传输前后的输出和输入端口,可以看出,机器 2 输出端口和机器 1 输入端口的 36 到 44 秒的时间窗口被占用,这个窗口大小正好是 8 秒,与计算结果一致。

对于 gdi1,它来自于任务 8,由任务 1 接收。任务 8 和任务 1 分别被分派到机器 1 和机器 0 上。从机器 1 到机器 0 的传输要使用机器 1 的输出端口和机器 0 的输入端口,但是对比 gdi1 传输前后端口的可用时间窗口可以看到, gdi1 的传输并未占用任何端口时间。这是因为任务 8 的输出由任务 3 进行了转送,任务 3 也被分派到了机器 0 上,这就使得从任务 3 转送任务 8 的输出数据到任务 1 时无需占用端口时间,加快了任务的整体完成时间。这时任务 3 输出(转送)的是任务 8 的输出量 Output[8],而不是其本身的输出量 Output[3]。

对于 gdi5,它来自于任务 1,由任务 6 接收。任务 1 和任务 6 被分别分派到机器 0 和机器 3 上。但

执行任务 4	执行任务 5
Out ports: 0: (0, 2147483647) 1: (0, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	Out ports: 0: (0, 2147483647) 1: (0, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)
In ports: 0: (0, 2147483647) 1: (0, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	In ports: 0: (0, 2147483647) 1: (0, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)
gdi3 : 36	gdi0 : 0
Out ports: 0: (0, 2147483647) 1: (0, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	Out ports: 0: (0, 2147483647) 1: (0, 7) (17, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)
In ports: 0: (0, 2147483647) 1: (0, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	In ports: 0: (0, 7) (17, 2147483647) 1: (0, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)
gdi2 : 7	gdi1 : 22
Out ports: 0: (0, 2147483647) 1: (0, 7) (17, 2147483647) 2: (0, 36) (44, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	Out ports: 0: (0, 2147483647) 1: (0, 7) (17, 2147483647) 2: (0, 36) (44, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)
In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)
gdi7 : 0	gdi6 : 57
Out ports: 0: (0, 2147483647) 1: (0, 7) (17, 52) (57, 2147483647) 2: (0, 36) (44, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	Out ports: 0: (0, 57) (93, 2147483647) 1: (0, 7) (17, 52) (57, 2147483647) 2: (0, 36) (44, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)
In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 52) (57, 2147483647)	In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 2147483647) 4: (0, 52) (93, 2147483647)
gdi4 : 0	gdi5 : 57
Out ports: 0: (0, 22) (93, 2147483647) 1: (0, 7) (17, 52) (57, 2147483647) 2: (0, 36) (44, 2147483647) 3: (0, 2147483647) 4: (0, 2147483647)	Out ports: 0: (0, 22) (93, 2147483647) 1: (0, 7) (17, 52) (57, 2147483647) 2: (0, 36) (44, 2147483647) 3: (0, 2147483647) 4: (0, 93) (105, 2147483647)
In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 22) (57, 2147483647) 4: (0, 52) (93, 2147483647)	In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 22) (57, 93) (105, 2147483647) 4: (0, 52) (93, 2147483647)
gdi9 : 98	gdi8 : 98
Out ports: 0: (0, 22) (93, 2147483647) 1: (0, 7) (17, 52) (57, 2147483647) 2: (0, 36) (44, 50) (52, 2147483647) 3: (0, 2147483647) 4: (0, 93) (105, 2147483647)	Out ports: 0: (0, 22) (93, 2147483647) 1: (0, 7) (17, 52) (57, 2147483647) 2: (0, 36) (44, 50) (52, 2147483647) 3: (0, 141) (168, 2147483647) 4: (0, 93) (105, 2147483647)
In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 22) (57, 93) (105, 2147483647) 4: (0, 50) (93, 2147483647)	In ports: 0: (0, 7) (17, 2147483647) 1: (0, 36) (44, 2147483647) 2: (0, 2147483647) 3: (0, 22) (57, 93) (105, 2147483647) 4: (0, 50) (93, 141) (168, 2147483647)

图 8 模型运行结果

对比 gdi5 传输前后的输出和输入端口,可以看出,机器 0 输出端口的可用时间窗口并无变化,而机器 4 输出端口的可用时间窗口却变了.这是因为这里使用了任务 7 对任务 1 的输出进行了转送.因为使用任务 7 进行任务转送会更早的完成数据传输.对于 gdi9,它来自于任务 4,由任务 0 接收.任务 4 和任务 0 分别被分派到了机器 2 和机器 4 上.因而其传输时间为:

$$gdi9 = \text{Output}[4] * \text{Comm.}[2][4] = 2$$

对比 gdi9 传输前后的输出和输入端口,可以看出,机器 2 输出端口和机器 4 输入端口的 50 到 52 秒的时间窗口被占用,这个窗口大小正好是 2 秒,与

计算结果一致.值得注意的是,目标机器时间为 98,而这里传输的完成时间是 52,传输在任务开始执行间已经完成,说明这里进行了数据的乱序调度.

对于 gdi8,由于 gdi9 进行了乱序调度,机器并未对 gdi9 的传输进行等待,因而在开始进行 gdi8 的传输时,目标机器时间依然是 98,这显示了乱序调度能减小任务群的执行完成时间.

gdi8、gdi7、gdi6、gdi4 的传输情况和 gdi0、gdi2 的传输情况比较类似,计算和分析方法也一样,这里不再赘述.

6 结束语

本文从异构体系运行时的任务分配、多核协同工作出发,在考虑了任务并行性的基础上,对通信数据进行了调度和优化,从而达到了更好的调度效果.

异构体系的计算能力还有很多挑战,如精确估计任务的运行时间、任务的自动划分和并行性的自动识别等.这些内容将在后续工作中进行研究.

参考文献:

- [1] Sarkar V. Partitioning and Scheduling Parallel Programs for Execution on MultiProcessors [D]. Starfprd; Stanford University, 1987.
- [2] Siegel H J, Antonio J K, Metzger R C, et al. Heterogeneous computing [M]. New York: McGraw-Hill, 1996:725-761.
- [3] Singh H, Youssef A. Mapping and scheduling heterogeneous task graphs using genetic algorithms [C]// Proc 1996 Heterogeneous Computing Workshop (HCW'96) IEEE Computer Society. Honolulu, HI: IEEE, 1996:86-97.
- [4] Lee Wang, Howard Jay Siegel. Task matching and scheduling in heterogeneous computing environments Using a genetic-algorithm-based approach[J]. Journal of Parallel and Distributed Computing, 1997(47): 8-22.
- [5] Maheswaran M, Ali S, Siegel H J, et al. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems [J]. Parallel Distribution Computer, 1999, 59(2): 107-121.
- [6] Cormen T H, Leiserson C E, Rivest R L. Introduction to Algorithms [M]. Cambridge, MA: MIT Press, 2001.
- [7] Donald E Knuth. 计算机程序设计艺术第 1 卷[M]. 3 版. 北京:国防工业出版社, 2007.

(下转第 153 页)

- Multimedia Technology. Piscataway, New Jersey: IEEE Press, 2011:227-230.
- [3] 王丹,高丰,朱鲁华. 基于遗传算法的 Fuzzing 测试用例生成模型[J]. 微电子学与计算机,2011,28(5):130-135.
- [4] Wang Xibo, Su Na. Automatic test data generation for path testing using genetic algorithms [C]//Proc of 2011 Third International Conference on Measuring Technology and Mechatronics Automation. Los Alamitos, CA: IEEE Computer Society Press, 2011: 596-599.
- [5] Zhong Cheng, Wu Xi-hua. Thread-level parallel generating inter-class test cases [C]//Proc of 4th International on Parallel Architectures, Algorithms and Programming. Los Alamitos, CA: IEEE Computer Society Press, 2011:42-46
- [6] 陈国良,王煦法,庄镇泉,等. 遗传算法及其应用[M]. 北京:人民邮电出版社,2001.
- [7] 多核系列教材编写组. 多核程序设计[M]. 北京:清华大学出版社,2007.

作者简介:

陈清媛 女,(1987-),硕士研究生. 研究方向为可信软件、网络与并行分布计算.

钟 诚 男,(1964-),博士,教授,博士生导师. 研究方向为可信软件、网络与并行分布计算.

李 智 男,(1967-),硕士,研究员. 研究方向为网络与并行分布计算.

(上接第 148 页)

- [8] 蒋建春,汪同庆. 一种异构多核处理器嵌入式实时操作系统构架设计[J]. 计算机科学,2011,38(6):298-303.

作者简介:

刘清涛 男,(1986-),硕士研究生. 研究方向为计算机体系结构.

于立新 男,(1967-),硕士,研究员. 研究方向为计算机体系结构、VLSI 和 SoC 设计.