中国科学 : 信息科学　2011 年　第 41 卷　第 3 期 : 349–364
www.scichina.com　info.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS
论　文

并行嵌入式系统中具有通信竞争任务调度问题的高级列表调度方法

穆鹏程[①*], NEZAN Jean-François[②], RAULET Mickaël[②], COUSIN Jean-Gabriel[②]

① 西安交通大学智能网络与网络安全教育部重点实验室, 电子与信息工程学院, 西安 710049
② IETR/Image and Remote Sensing Group, CNRS UMR 6164/INSA Rennes, 35043 RENNES Cedex, France
* 通信作者. E-mail: pengchengmu@gmail.com

收稿日期: 2009–09–19; 接受日期: 2010–04–15
国家自然科学基金 (批准号: 60971113, 61071125) 和中国博士后科学基金 (批准号: 20100481346) 资助项目

摘要　现代嵌入式系统正在向使用多核或多处理器进行并行处理的方向发展. 针对并行嵌入式系统中具有通信竞争情况下的任务调度问题, 文中提出 3 项高级技术以提高列表调度方法的性能. 首先使用 5 组 (已存在的两组和新提出的 3 组) 节点等级作为节点优先权来生成节点列表; 然后使用关键子节点技术来改善调度过程中处理器的选择; 最后使用通信延迟技术扩大通信连接线上的空闲时间区间. 文中还给出了组合使用这 3 项技术的高级动态列表调度方法. 实验结果表明, 在中等通信代价和高通信代价的情况下, 组合高级动态方法能够有效地缩短调度长度, 可以在通信代价很高时通过优化使用硬件资源使调度结果加速高达 80%.

关键词　列表调度　通信竞争　节点等级　关键子节点　通信延迟

1　引言

数字通信和视频压缩等技术的最新发展极大地提高了算法和系统的复杂度, 为此, 使用多核以及多硬件加速器的片上系统 (SoC, system on a chip) 正在成为构建复杂系统的基本要素, 而数据流编程则用于多处理器编程之中 [1]. 随着对应用程序执行效率要求的提高, 数据流程序在嵌入式系统中多个器件之间的任务调度也越来越重要. 任务调度过程一般非常复杂, 人工处理时往往得到的是次优解. 通用并行计算机结构中的任务调度问题已经被广泛研究, 并行嵌入式系统 [2] 中的任务调度问题与之稍有不同, 核与核之间的通信对于任务的调度以及硬件资源的使用有很大影响. 因此, 我们需要针对并行嵌入式系统研究新的任务调度方法以获得最优或近似最优解.

在研究任务调度问题时, 应用程序通常被建模成有向无环的任务图 (DAG, directed acyclic graph)[2,3], 其中节点代表任务 (运算), 边代表任务之间的数据流 (通信). 任务调度的目的是把运算和通信分别分配到目标系统中的处理器和总线 (通信连接线) 上以得到最小的调度长度 (makespan). 调度可以是静态的 (编译时进行) 也可以是动态的 (运行时进行). 嵌入式系统中的应用程序往往具有

引用格式: 穆鹏程, Nezan J F, Raulet M, 等. 并行嵌入式系统中具有通信竞争任务调度问题的高级列表调度方法. 中国科学: 信息科学, 2011, 41: 349–364

确定性, 此时静态调度能够减少源代码以提高运算效率, 因此比动态调度更加适用. 本文研究并行嵌入式系统编程时的静态调度问题, 所有任务调度方法都是在编译时使用.

一般性的任务调度已经被证明是 NP 难题 [3,4], 因此, 许多文章研究启发式方法以期得到近似最优解. 早期的任务调度方法不考虑任务间的通信影响 [5,6], 随着通信量在现代应用中的增加, 许多任务调度方法开始考虑通信 [3,7−10]. 这些方法绝大多数使用完全连接的系统拓扑结构, 所有的通信可以同时进行. 近年来, 任意连接的处理器网络结构被用来更准确地描述并行系统 [11−15], 任务调度也开始考虑通信竞争问题.

以上方法大多都是基于列表调度方法, 该方法在有通信竞争时的基本技术已经在文献 [16] 中给出. 本文将针对并行嵌入式系统中有通信竞争的任务调度问题给出高级列表调度方法. 首先, 在已有的两组节点等级的基础上定义 3 组新的节点等级, 它们将作为节点优先权用来生成节点列表; 其次, 使用关键子节点技术改善处理器选择的效果; 第三, 通信延迟技术在必要时还可以扩大通信连接线上的空闲时间区间. 本文最后把这些技术组合在一起以改善任务调度的结果.

本文组织如下: 第 2 节首先给出本文所用到的模型和定义, 然后介绍有通信竞争的任务调度问题; 各种考虑通信竞争的节点等级将在第 3 节给出; 第 4 节将给出详细的列表调度方法; 第 5 节通过实验来比较本文提出的高级方法与经典方法的不同效果; 第 6 节总结全文.

2 模型与定义

本文把要调度的程序称为应用算法 (algorithm) 并用 DAG 模型来表示. 具有多处理器的并行嵌入式系统称为系统结构 (architecture) 并用拓扑图模型来表示. 下面将给出这两个模型的详细介绍.

2.1 DAG 模型

DAG 是一个有向无环图, 记为 $G = (V, E, w, c)$, 其中 V 是所有节点的集合, E 是所有边的集合. 给定两个节点 $n_i, n_j \in V$, e_{ij} 表示从起点 n_i 到终点 n_j 的边. 节点代表运算, 节点 n_i 的权重 $w(n_i)$ 代表运算的时间代价; 边代表两个节点之间的通信, 边 e_{ij} 的权重 $c(e_{ij})$ 代表通信的时间代价. 节点 n_i 的所有直接前趋构成集合 $\mathrm{pred}(n_i) = \{n_x \in V : e_{xi} \in E\}$, 所有直接后继构成集合 $\mathrm{succ}(n_i) = \{n_x \in V : e_{ix} \in E\}$. 满足 $\mathrm{pred}(n_i) = \emptyset$ 的节点称为源点, 满足 $\mathrm{succ}(n_i) = \emptyset$ 的节点称为汇点, 其中 $\emptyset$ 表示空集.

运算在处理器上串行执行, 其作为一个整体不能被拆分成多个部分. 运算的执行必须等所有输入通信结束后才能开始; 而所有输出通信必须等运算结束后才能开始. 同一个通信连接线上的所有通信也是串行的, 但是在满足上述输入输出限制的情况下, 运算和通信可以并行进行. 图 1(a) 给出了一个应用算法的 DAG 模型, 它在文献 [17] 中被用来比较不同调度方法的性能, 本文 5.1 小节也将用到它.

2.2 拓扑图模型

拓扑图 $\mathrm{TG} = (N, P, D, H, b)$ 被用来建模由通信连接线和交换器组成的多处理器系统 [14]. 这里 N 是所有顶点的集合; P 是 N 的子集, $P \subseteq N$; D 是有向边的集合; H 是超边的集合; b 是边的相对数据速率. D 和 H 的合集称为连接线集合 L, $L = D \cup H$, 该集合的元素记为 $l \in L$, 拓扑图也可以记为 $\mathrm{TG} = (N, P, L, b)$.

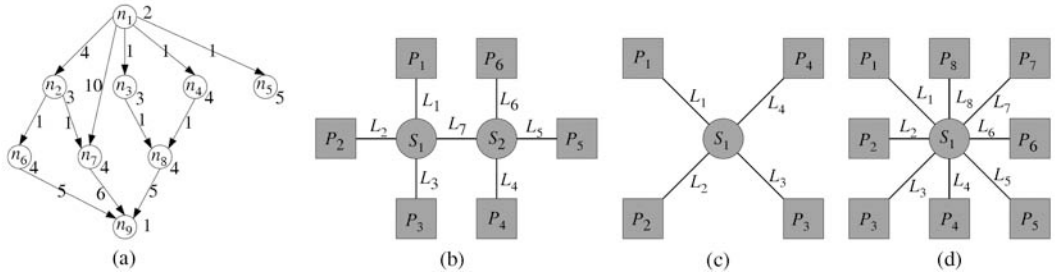

图 1 系统模型

Figure 1 System models

(a) Algorithm; (b) architecture 1; (c) architecture 2; (d) architecture 3

由于并行嵌入式系统通常使用多种异构器件, 因此本文使用拓扑图对其进行建模. 对于拓扑图 $\mathrm{TG} = (N, P, L, b)$, 顶点 $p \in P$ 代表处理器, 顶点 $n \in N, n \notin P$ 代表交换器. 假设暂不使用有向边, 则连接线 $l \in L$ 实际上是一个超边 h, 它是 N 的一个包含多个顶点的子集, $h \subseteq N, |h| > 1$. 连接多个顶点的超边代表半双工的多方向通信连接线 (如总线). 连接线 $l \in L$ 的权重 $b(l)$ 代表 l 的相对数据速率.

与处理器不同, 交换器是一种只用来连接通信连接线的理想顶点, 任何运算不能在交换器上运行.

理想交换器 对于交换器 s, 假定 $l_1, l_2, \ldots, l_n$ 是连接在它上面的所有通信连接线. 如果其中的 l_{i_1} 和 l_{i_2} 空闲, 那么通信可以从 l_{i_1} 传到 l_{i_2} 而不受连接于 s 的其他连接线上通信的影响, 同时也不会对其他连接线上的通信造成影响.

图 1(b) 给出了一个由 7 条通信连接线 (L_1, L_2, L_3, L_4, L_5, L_6, L_7) 和 2 个交换器 (S_1, S_2) 连接起来的具有 6 个处理器 (P_1, P_2, P_3, P_4, P_5, P_6) 的系统结构, 它描述了 TI 的包含两个 C6474 多核数字信号处理器的评估板 [1)]. 图 1(c), (d) 还给出了另外两种系统结构, 它们将分别用于 5.1 和 5.2 小节的实验验证中.

并行嵌入式系统使用链路从一个处理器到另一个处理器传输数据. 链路由一串从起点处理器到终点处理器的通信连接线组成, 它们之间通过交换器连接. 举例来说, 在图 1(b) 中 $L_1 \to L_7 \to L_4$ 是一条从 P_1 到 P_4 的链路. 链路 R 上的连接线 l 记为 $l \in R$, 从处理器 p_i 到处理器 p_j 的所有链路构成集合 $\mathrm{RS}(p_i, p_j)$, 并且当 $p_i = p_j$ 时有 $\mathrm{RS}(p_i, p_j) = \emptyset$, 即不需要任何链路.

路由是用来生成通信链路的过程, 它是任务调度的一个重要方面. 文献 [15] 在调度中动态生成链路以改善性能, 但是其系统结构中没有使用交换器. 事实上, 使用交换器的并行嵌入式系统中的链路可以一次性确定好后放在存储表中, 即使用静态链路. 本文将使用这种静态链路, 并假定任何两个处理器之间至少存在一条链路, 此时任务调度中的路由问题变为查找存储表.

2.3 具有通信竞争的任务调度

调度一个 DAG 指的是给 DAG 中每个节点分配一个处理器并赋予一个开始时刻. 当考虑通信竞争时, 调度还需要把每个通信分配到连接线上并赋予开始时刻. 在拓扑图 $\mathrm{TG} = (N, P, L, b)$ 上对 DAG $G = (V, E, w, c)$ 的调度 S 可以描述如下.

1) http://www.ti.com/

节点 $n_i \in V$ 在处理器 $p \in P$ 上的开始时刻记为 $t_s(n_i,p)$, 结束时刻为 $t_f(n_i,p)=t_s(n_i,p)+w(n_i,p)$, 其中 $w(n_i,p)$ 是 n_i 在 p 上的运行时间. S 的调度长度指所有节点的最晚结束时刻, 即

$$sl(S)=\max_{n_i\in V}\{t_f(n_i,\mathrm{proc}(n_i))\},$$

其中 $\mathrm{proc}(n_i)$ 表示分配给 n_i 的处理器.

由于节点在不同处理器上的运行时间会差别很大 ($w(n_i,p_j)\gg w(n_i,p_k)$), 因此往往被限定在运行时间相对较短的处理器上, 能够运行 n_i 的处理器集合记为 $\mathrm{Proc}(n_i)$. 一个节点在不同处理器上的平均运算时间代表该节点的权重, 即 $w(n_i)=\frac{1}{|\mathrm{Proc}(n_i)|}\sum_{p\in\mathrm{Proc}(n_i)}w(n_i,p)$, 其中 $|\mathrm{Proc}(n_i)|$ 表示 $\mathrm{Proc}(n_i)$ 中的处理器个数.

只有当边的起点和终点分配在不同处理器上时通信才进行, 边 $e_{ij}\in E$ 在 $l\in R$ 上的开始时刻记为 $t_s(e_{ij},l,R)$. 由于嵌入式系统通常使用 cut-through 模式的电路交换进行通信, 因此 e_{ij} 在链路 $R=l_1\to l_2\to\cdots\to l_k$ 的所有连接线上对齐, 即 $t_s(e_{ij},l_1,R)=t_s(e_{ij},l_2,R)=\cdots=t_s(e_{ij},l_k,R)$. 记分配给 e_{ij} 的链路为 $R(e_{ij})$, e_{ij} 在 $R=R(e_{ij})$ 的所有连接线上的开始和结束时刻分别记为 $t_s(e_{ij},R)$ 和 $t_f(e_{ij},R)$, 并有 $t_f(e_{ij},R)=t_s(e_{ij},R)+\frac{d(e_{ij})}{\min_{l\in R}\{b(l)\}}$, 其中 $d(e_{ij})$ 是 e_{ij} 传输的数据量, $\min_{l\in R}\{b(l)\}$ 是 R 中各连接线的最小数据速率. 边的权重为其在所有可能链路上的平均通信时间, 即 $c(e_{ij})=\frac{1}{\sum_{p_x,p_y}|\mathrm{RS}(p_x,p_y)|}\sum_{p_x,p_y}\{\sum_{R\in\mathrm{RS}(p_x,p_y)}\frac{d(e_{ij})}{\min_{l\in R}\{b(l)\}}\}$, 其中 $p_x\in\mathrm{Proc}(n_i),p_y\in\mathrm{Proc}(n_j)$. 这种计算 $c(e_{ij})$ 的方法是本文首次提出的, 它更适合于并行嵌入式系统中的任务调度.

当节点的所有输入通信结束时, 该节点才可以开始在处理器上进行运算, 该时刻称为数据就绪时刻 (DRT, data ready time), 表示为 $\mathrm{DRT}(n_j,p)=\max_{e_{ij}\in E}\{t_f(e_{ij},R(e_{ij}))\}$. DRT 是一个节点可以开始的最早时刻, 若节点 n_j 没有输入边, 那么 $\mathrm{DRT}(n_j,p)=0,\forall p\in P$, 这表示 n_j 的数据在最开始 (0 时刻) 已经准备就绪.

对节点和边进行调度时通常使用插入技术 (insertion technique)[16], 该技术需要满足以下条件.

节点调度条件 假定 $[A,B](A,B\in[0,\infty])$ 是处理器 p 上的一个空闲时间区间, 则当 $\max\{A,\mathrm{DRT}(n_i,p)\}+w(n_i,p)\leqslant B$ 时 n_i 可以安排在 p 上的 $[A,B]$ 区间内. 此时 n_i 在 p 上的开始时刻为 $t_s(n_i,p)=\max\{A,\mathrm{DRT}(n_i,p)\}$.

边调度条件 假定 R 是传输边 e_{ij} 的链路并且 $[A,B](A,B\in[0,\infty])$ 是该链路中所有连接线上的一个公共空闲时间区间, 则当 $\max\{A,t_f(n_i,\mathrm{proc}(n_i))\}+\frac{d(e_{ij})}{\min_{l\in R}\{b(l)\}}\leqslant B$ 时 e_{ij} 可以被安排在 R 上的 $[A,B]$ 区间内, 此时 e_{ij} 在该链路上的开始时刻为 $t_s(e_{ij},R)=\max\{A,t_f(n_i,\mathrm{proc}(n_i))\}$.

3 考虑通信竞争的节点等级

节点优先权 (node priority) 对于 DAG 调度非常重要, 节点等级 top level 和 bottom level 通常被用作节点优先权 [11,18]. 节点的 top level 是指从任意一个源点到该节点的最长路径的长度, 但不包括该节点的权重; 节点的 bottom level 是指从该节点到任意一个汇点的最长路径的长度, 并且包括该节点的权重. 任务调度方法中常用的节点等级有两组: 1) computation top level 和 computation bottom level ($\mathrm{tl}_{\mathrm{comp}}$ 和 $\mathrm{bl}_{\mathrm{comp}}$), 2) top level 和 bottom level (tl 和 bl). 本文将给出 3 组新的节点等级: 3) input top level 和 input bottom level ($\mathrm{tl}_{\mathrm{in}}$ 和 $\mathrm{bl}_{\mathrm{in}}$), 4) output top level 和 output bottom level ($\mathrm{tl}_{\mathrm{out}}$ 和 $\mathrm{bl}_{\mathrm{out}}$), 5) input/output top level 和 input/output bottom level ($\mathrm{tl}_{\mathrm{io}}$ 和 $\mathrm{bl}_{\mathrm{io}}$). 图 2 给出了不同的 top level 和

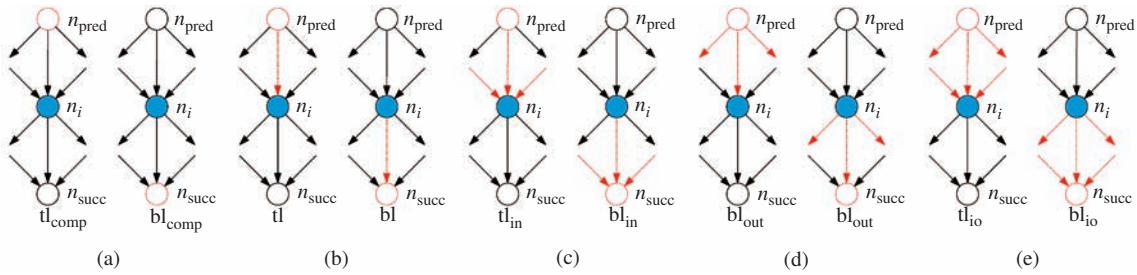

图 2 五组节点等级

Figure 2 Five groups of node levels

bottom level 中节点间的依赖关系, 其中红色虚线节点和边被用来定义 n_i 的各种 top level 和 bottom level.

3.1 computation top level 和 computation bottom level (图 2(a))

节点的 computation top level 是指从任意一个源节点到该节点的只包括节点权重的最长路径的长度; 它的 computation bottom level 是指从该节点出发到任意一个汇点的只包括节点权重的最长路径的长度. 在定义 computation top level 和 computation bottom level 时不考虑边的权重, 它们可以用下面的公式递归定义:

$$\text{tl}_{\text{comp}}(n_i) = \begin{cases} 0, & \text{如果 } n_i \text{ 是一个源点,} \\ \max\limits_{n_k \in \text{pred}(n_i)} \{\text{tl}_{\text{comp}}(n_k) + w(n_k)\}, & \text{其他情况,} \end{cases}$$

$$\text{bl}_{\text{comp}}(n_i) = \begin{cases} w(n_i), & \text{如果 } n_i \text{ 是一个汇点,} \\ \max\limits_{n_k \in \text{succ}(n_i)} \{\text{bl}_{\text{comp}}(n_k)\} + w(n_i), & \text{其他情况.} \end{cases}$$

3.2 top level 和 bottom level (图 2(b))

与 computation top level 和 computation bottom level 相比, top level 和 bottom level 额外考虑了路径上边的权重, 它们可以用下面的公式递归定义:

$$\text{tl}(n_i) = \begin{cases} 0, & \text{如果 } n_i \text{ 是一个源点,} \\ \max\limits_{n_k \in \text{pred}(n_i)} \{\text{tl}(n_k) + w(n_k) + c(e_{ki})\}, & \text{其他情况.} \end{cases}$$

$$\text{bl}(n_i) = \begin{cases} w(n_i), & \text{如果 } n_i \text{ 是一个汇点,} \\ \max\limits_{n_k \in \text{succ}(n_i)} \{\text{bl}(n_k) + c(e_{ik})\} + w(n_i), & \text{其他情况.} \end{cases}$$

3.3 input top level 和 input bottom level (图 2(c))

input top level 和 input bottom level 不仅考虑了路径上节点的权重, 还考虑了路径上节点的所有输入边的权重, 它们可以用下面的公式递归定义:

$$\mathrm{tl}_{\mathrm{in}}(n_i)=\begin{cases}0, & \text{如果 } n_i \text{ 是一个源点,}\\ \max\limits_{n_k\in\mathrm{pred}(n_i)}\{\mathrm{tl}_{\mathrm{in}}(n_k)+w(n_k)\}+\sum\limits_{e_{li}\in E}c(e_{li}), & \text{其他情况.}\end{cases}$$

$$\mathrm{bl}_{\mathrm{in}}(n_i)=\begin{cases}w(n_i), & \text{如果 } n_i \text{ 是一个汇点,}\\ \max\limits_{n_k\in\mathrm{succ}(n_i)}\{\mathrm{bl}_{\mathrm{in}}(n_k)+\sum\limits_{e_{lk}\in E}c(e_{lk})\}+w(n_i), & \text{其他情况.}\end{cases}$$

3.4 output top level 和 output bottom level (图 2(d))

output top level 和 output bottom level 不仅考虑了路径上节点的权重, 还考虑了路径上节点的所有输出边的权重, 它们可以用下面的公式递归定义:

$$\mathrm{tl}_{\mathrm{out}}(n_i)=\begin{cases}0, & \text{如果 } n_i \text{ 是一个源点,}\\ \max\limits_{n_k\in\mathrm{pred}(n_i)}\{\mathrm{tl}_{\mathrm{out}}(n_k)+w(n_k)+\sum\limits_{e_{kl}\in E}c(e_{kl})\}, & \text{其他情况.}\end{cases}$$

$$\mathrm{bl}_{\mathrm{out}}(n_i)=\begin{cases}w(n_i), & \text{如果 } n_i \text{ 是一个汇点,}\\ \max\limits_{n_k\in\mathrm{succ}(n_i)}\{\mathrm{bl}_{\mathrm{out}}(n_k)\}+\sum\limits_{e_{il}\in E}c(e_{il})+w(n_i), & \text{其他情况.}\end{cases}$$

3.5 input/output top level 和 input/output bottom level (图 2(e))

input/output top level 和 input/output bottom level 不仅考虑了路径上节点的权重, 还考虑了路径上节点的所有输入边和输出边的权重, 它们可以用下面的公式递归定义:

$$\mathrm{tl}_{\mathrm{io}}(n_i)=\begin{cases}0, & \text{如果 } n_i \text{ 是一个源点,}\\ \max\limits_{n_k\in\mathrm{pred}(n_i)}\{\mathrm{tl}_{\mathrm{io}}(n_k)+w(n_k)+\sum\limits_{e_{kl}\in E}c(e_{kl})-c(e_{ki})\}+\sum\limits_{e_{li}\in E}c(e_{li}), & \text{其他情况,}\end{cases}$$

$$\mathrm{bl}_{\mathrm{io}}(n_i)=\begin{cases}w(n_i), & \text{如果 } n_i \text{ 是一个汇点,}\\ \max\limits_{n_k\in\mathrm{succ}(n_i)}\{\mathrm{bl}_{\mathrm{io}}(n_k)+\sum\limits_{e_{lk}\in E}c(e_{lk})-c(e_{ik})\}+\sum\limits_{e_{il}\in E}c(e_{il})+w(n_i), & \text{其他情况.}\end{cases}$$

已存在的两组节点等级通常用在无通信竞争的列表调度中, 与之相比, 3 组新的节点等级则充分考虑了通信竞争. 表 1 给出了图 1(a) 中所示 DAG 的所有五组节点等级, 该表将在 5.1 小节中使用.

该算法主要由 3 部分组成, 首先通过 Sort_Nodes() 把节点排成一个静态列表, 然后依次为每个节点选择处理器 (Select_Processor()) 和进行任务调度 (Schedule_Node()). 由于列表中节点的次序会影响调度结果, 人们提出了多种优先权策略 [10,11]. 实验表明, 在有通信竞争的情况下使用 bottom level 作为优先权进行节点排序的列表调度方法优于其他方法 [18], 因此本文使用如下静态节点排序准则.

静态节点排序准则 按照节点的 bottom level 进行降序排列; 如果两个节点具有相同的 bottom level, 那么 top level 大的节点排在前面; 如果 bottom level 和 top level 都相同, 则随机排列各节点.

表 1 不同的节点等级

Table 1 Different node levels

	$\mathrm{tl_{comp}}$	$\mathrm{bl_{comp}}$	tl	bl	$\mathrm{tl_{in}}$	$\mathrm{bl_{in}}$	$\mathrm{tl_{out}}$	$\mathrm{bl_{out}}$	$\mathrm{tl_{io}}$	$\mathrm{bl_{io}}$
n_1	0	11	0	23	0	41	0	35	0	55
n_2	2	8	6	15	6	35	19	16	19	36
n_3	2	8	3	14	3	26	19	14	19	26
n_4	2	9	3	15	3	27	19	15	19	27
n_5	2	5	3	5	3	5	19	5	19	5
n_6	5	5	10	10	10	21	24	10	24	21
n_7	5	5	12	11	20	21	24	11	34	21
n_8	6	5	8	10	9	21	24	10	25	21
n_9	10	1	22	1	40	1	34	1	54	1

4 列表调度方法

列表调度是一种重要的任务调度方法, 算法 1 给出了一种常用的静态列表调度方法 [14].

算法 1: Static_List_Scheduling (G, TG)

输入: DAG $G = (V, E, w, c)$ 和拓扑图 TG=(N, P, L, b)

输出: G 在 TG 上的调度

```
1  NodeList ← Sort_Nodes(V);
2  for 每一个 n ∈ NodeList do
3      p_best ← Select_Processor(n, P);
4      Schedule_Node(n, p_best);
5  end
```

静态列表调度方法的详细介绍参见文献 [16], 本文把该方法称为经典列表调度方法, 并用它跟我们的高级方法进行比较. 下面将介绍两项高级列表调度技术以及高级动态列表调度方法.

4.1 使用关键子节点技术进行处理器选择

经典的列表调度方法为节点选择能够提供最早完成时刻的处理器, 该准则往往会造成局部最优结果. 事实上, 该准则用于汇合结构的 DAG 时经常会得到较差的结果, 尤其是在通信代价很高并且具有竞争的情况下. 图 3(a) 给出了一个这样的例子, 图 3(b) 给出了经典的处理器选择方法所得到的调度结果, 该方法为 n_1, n_2 和 n_3 分别选择一个新的处理器以保证它们能够尽早完成. 此时 n_4 必须等来自 n_2 和 n_3 的通信结束后才能开始运行, 其最终调度长度为 6. 相比之下, 图 3(c) 中把所有节点安排到同一个处理器的调度长度则是 4. 造成上述结果的原因主要是没有充分考虑后继节点对选择处理器的影响, 为此本文提出一种关键子节点技术.

关键子节点在文献 [10] 中定义为一个节点的后继节点中绝对最晚开始时刻 (absolute latest possible start time, ALST) 和绝对最早开始时刻 (absolute earliest possible start time, AEST) 的差最小的节点. 这种关键子节点被用在不限定处理器个数并且无通信竞争的任务调度中, 本文将在限定处理器个数的情况下使用关键子节点的概念研究具有通信竞争的列表调度, 关键子节点定义如下.

关键子节点 给定节点列表 NodeList, 节点 n_i 的关键子节点记为 $cc(n_i)$, 它是 n_i 的后继节点中第一个出现在 NodeList 中的节点.

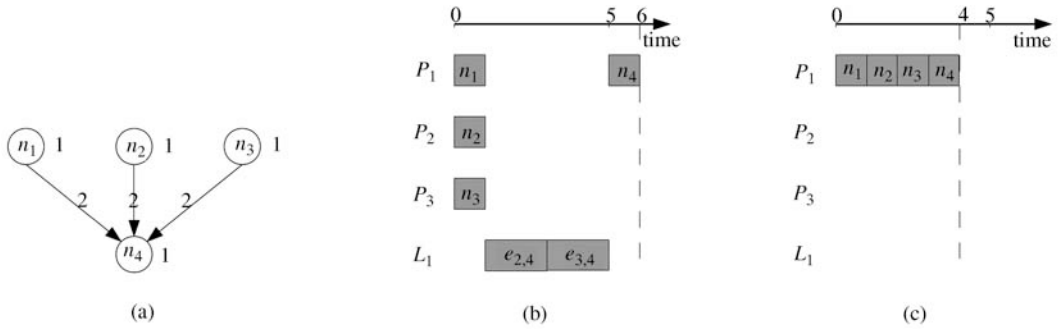

图 3　汇合结构的 DAG 和两个不同的调度结果

Figure 3　A join DAG and two different schedule results

根据上述定义, 即使 DAG 不变, 当 NodeList 不相同时, n_i 的关键子节点也可能会不同, 这是与文献 [10] 中关键子节点的最大区别之处. 在选择处理器时使用关键子节点不仅要考虑节点的前趋, 而且还要考虑节点的最主要后继, 算法 2 给出了使用关键子节点进行处理器选择的方法.

算法 2: Select_Processor (n_i, P)

输入: 节点 $n_i \in V$ 和所有处理器的集合 P

输出: 运行输入节点 n_i 的最佳处理器 p_{best}

```
1  寻找关键子节点 cc(n_i);
2  BestFinishTime ← ∞;
3  for 每一个 p ∈ Proc(n_i) do
4     FinishTime ← Schedule_Node (n_i, p, true);
5     MinFinishTime ← ∞;
6     if cc(n_i) ≠ null then
7        for 每一个 p' ∈ Proc(cc(n_i)) do
8           FinishTime ←
            Schedule_Node(cc(n_i), p', true);
9           MinFinishTime ← min{MinFinishTime,FinishTime};
10          取消对 cc(n_i) 的输入边的调度;
11          取消 cc(n_i) 在 p' 上的调度;
12       end
13    else
14       MinFinishTime ← FinishTime;
15    end
16    if MinFinishTime<BestFinishTime then
17       BestFinishTime ← MinFinishTime;
18       p_best ← p;
19    end
20       取消对 n_i 的输入边的调度;
21       取消 n_i 在 p 上的调度;
22    end
```

如果一个未调度节点的所有前趋都已经被调度过, 那么该节点称为自由节点. 由于在对 n_i 进行处理器选择时 $cc(n_i)$ 很可能不是自由节点, 因此在 Select_Processor() 中对 $cc(n_i)$ 调度时只考虑关键子节点的已经完成调度的前趋, 这一点将会体现在边的调度算法中.

4.2　使用通信延迟进行节点和边调度

本文中节点和边的调度方法与经典方法的不同之处在于使用尽可能晚 (as late ss possible, ALAP) 的开始时间进行通信延迟. 给定边 e_{ij} 和它所处的链路 $R = l_1 \to l_2 \to \cdots \to l_k$, 设 e_m 是连接线 l_m 上紧随 e_{ij} 之后的边, e_{ij} 的 ALAP 定义为 $\text{ALAP}(e_{ij}) = \min\{t_s(e_1, R(e_1)), t_s(e_2, R(e_2)), \ldots, t_s(e_k, R(e_k)), t_s(n_j, \text{proc}(n_j))\} - \frac{d(e_{ij})}{\min_{l \in R}\{b(l)\}}$. 如果 e_m 不存在 (即 e_{ij} 是连接线 l_m 上的最后一条边), 那么可以认为 $t_s(e_m, R(e_m)) = \infty$.

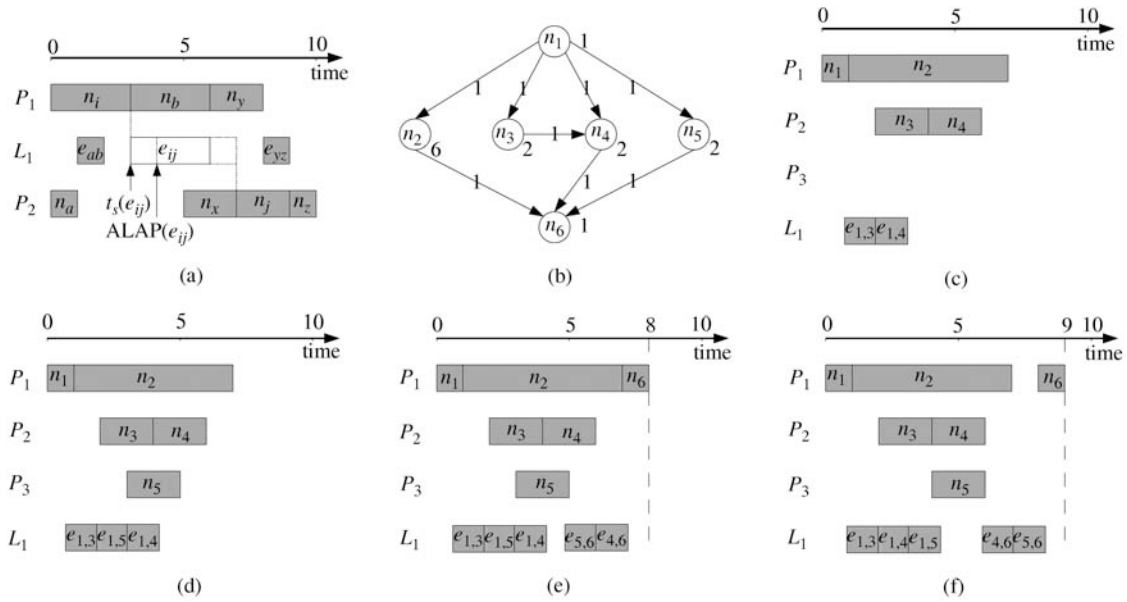

图 4 通信延迟

Figure 4 Communication delay

(a) ALAP; (b) a DAG example; (c) partial schedule result; (d) schedule n_5 with communication delay; (e) schedule result with communication delay; (f) schedule result without communication delay

ALAP 可以用来延迟通信, 此时连接线上的空闲时间区间被扩大. l 上相继两条边 e_{n-1} 和 e_n 之间的空闲时间区间可以从 $[t_f(e_{n-1}, R(e_{n-1})), t_s(e_n, R(e_n))]$ 扩大为 $[t_f(e_{n-1}, R(e_{n-1})), \text{ALAP}(e_n)]$. 如果 e_n 是 l 上的第一条边, 那么 $t_f(e_{n-1}, R(e_{n-1})) = 0$; 如果 e_{n-1} 是 l 上的最后一条边, 那么 $t_s(e_n, R(e_n)) = \text{ALAP}(e_n) = \infty$.

图 4(a) 给出了如何使用 ALAP 的示意图, 如果 e_{ij} 被延迟到 ALAP, 那么 L_1 上 e_{ab} 和 e_{ij} 之间的空闲时间区间将会扩大, 于是数据量更大的通信就可以插入到 e_{ab} 与 e_{ij} 之间.

算法 3 给出了在处理器 p 上调度节点 n_i 的方法, 其中节点输入边 ALAP 的计算必须在该节点被真正调度后进行 (算法 3 中第 6 到 10 行), 不能在选择处理器的过程中计算 ALAP. 因此, 我们使用了一个布尔值来指示是否在 Select_Processor() 中调用 Schedule_Node().

算法 3: Schedule_Node (n_i, p, IsTemporary)

输入: 节点 $n_i \in V$, 处理器 $p \in \text{Proc}(n_i)$ 和布尔值 IsTemporary

输出: n_i 在 p 上的结束时刻

1 for 每一个 $n_l \in \text{pred}(n_i), \text{proc}(n_l) \neq p$ do
2 Schedule_Edge(e_{li}, p);
3 end
4 计算 n_i 的 DRT;
5 根据节点调度条件在 p 上为 n_i 寻找最早的空闲时间区间;
6 if IsTemporary = false then
7 for 每一个 $n_l \in \text{pred}(n_i), \text{proc}(n_l) \neq p$ do
8 计算 e_{li} 的 ALAP;
9 end
10 end
11 在 p 上调度 n_i 并计算其结束时刻;

算法 4 给出了边的调度方法, 尽管与经典方法相似, 但是仍然有一些改进. 由于关键子节点的某些前趋可能还没有被调度过, 我们要验证 e_{ij} 的起点 n_i, 此外我们选择最佳链路以保证通信尽可能早

完成, 并且在边调度条件中还使用 ALAP 进行通信延迟.

算法 4: Schedule_Edge(e_{ij}, p)

输入: 边 $e_{ij} \in E$ 和安排给 n_j 的处理器 $p \in \text{Proc}(n_j)$
输出: 空

```
1  if n_i 已经被调度过 then
2    if proc(n_i) ≠ p then
3      FinishTime ← ∞;
4      for 每一个 R ∈ RS(proc(n_i), p) do
5        在 R 的所有连接线上寻找最早的
         公共空闲时间区间, 使其满足使
         用 ALAP 延迟的边调度条件;
6        if t_f(e_ij, R) <FinishTime then
7          FinishTime← t_f(e_ij, R);
8          R_best ← R;
9        end
10     end
11     在 R_best 上调度 e_ij;
12   end
13 end
```

图 4(a) 给出了一个 DAG 实例来说明使用通信延迟的效果. 通过使用优先权 bl & tl, 节点被排列成 n_1, n_2, n_3, n_4, n_5, n_6 的静态列表. 图 4(c) 给出了调度完 n_1, n_2, n_3, n_4 后的部分调度结果. 当调度 n_5 时, n_4 的输入边 $e_{1,4}$ 可以延迟到它的 ALAP(时刻 3), 因此 $e_{1,5}$ 可以被插入 $e_{1,3}$ 和 $e_{1,4}$ 之间, 如图 4(d) 所示, 最终得到图 4(e) 中长度为 8 的调度结果. 如果不使用 ALAP, 调度结果将如图 4(f) 所示, 长度为 9.

4.3 高级动态列表调度

算法 5 给出了一种新的高级动态列表调度方法, 这里 "动态" 指节点列表不是在调度之前确定, 而是在调度的过程中生成. 静态列表调度中的 Sort_Nodes() 已不再需要, 取而代之的是 Choose_Node(). Select_Processor() 和 Schedule_Node() 则使用上面给出的新方法.

算法 5: Dynamic_List_Scheduling (G, TG)

输入: DAG $G = (V, E, w, c)$ 和拓扑图 TG= (N, P, L, b)
输出: G 在 TG 上的调度

```
1  UNS ← V
2  while UNS 不空 do
3    n ← Choose_Node (UNS);
4    p_best ← Select_Processor (n, P);
5    Schedule_Node(n, p_best, false);
6    从 UNS 中去掉 n;
7  end
```

与静态节点列表相同, 节点等级对于生成动态节点列表仍然有效. 在动态列表调度中, 尽管任意一个自由节点都可以在下一步中进行调度, 但是我们应当选择最关键的节点优先处理. 在调度过程中包含自由节点的最长路径的长度对于最终的调度长度至关重要, 该路径上的自由节点 (即该路径上第一个没有被调度的节点) 应该在下一步中被立即调度以保证其能够尽可能早地完成. 该自由节点称为关键节点, 它可以通过考查 bottom level 得到, 具体做法如算法 6 所示.

该算法中使用 bottom level($\text{bl}(n_i)$) 作为节点优先权. bottom level 反映了从节点开始到 DAG 结束所需要的时间, 而本文提出的新的 bottom level 能更好地反映通信竞争时的情况, 因此可以用 $\text{bl}_{\text{comp}}(n_i)$, $\text{bl}_{\text{in}}(n_i)$, $\text{bl}_{\text{out}}(n_i)$ 和 $\text{bl}_{\text{io}}(n_i)$ 等 bottom level 代替 $\text{bl}(n_i)$. 使用不同的 bottom level 可能会形成不同的动态节点列表, 从而最终会得到不同的调度结果.

算法 6: Choose_Node (UN)

输入: 由所有未调度节点组成的集合 UN

输出: 所有未调度节点中的关键节点 n_c

```
从 UN 中找出所有的自由节点组成集合 FN;
MaxLength ← 0;
for 每一个 n_i ∈ FN do
    Length ← 0;
    for 每一个 n_l ∈ pred(n_i) do
        Length ← max{Length, t_f(n_l, proc(n_l)) + bl(n_i)};
    end
    if MaxLength<Length then
        MaxLength← Length;
        n_c ← n_i;
    else if MaxLength = Length then
        if bl(n_c) < bl(n_i) then
            n_c ← n_i;
        end
    end
end
```

5 实验结果

本节给出上述高级列表调度方法同文献 [14] 中经典列表调度方法相比较的实验结果, 图 1(c), (d) 中的系统结构分别用在 5.1 和 5.2 小节.

5.1 实例比较

我们使用图 1(a) 中所给出的 DAG 来展示高级动态列表调度方法以及不同的节点优先权对调度结果的改进. 该 DAG 的所有 5 组 top level 和 bottom level 已经在表 1 中给出, 表 2 中给出根据节点排序准则得到的静态列表, 该表同时还给出根据不同的静态列表得到的关键子节点.

图 5 给出了使用 bl & tl 作为节点优先权的经典静态列表调度方法所得到的调度结果, 我们对同一条边使用两种符号分别表示发送和接收, 其最终调度长度为 21.

在我们的高级动态列表调度方法中使用不同的节点优先权可能会生成不同的动态节点列表, 并最终得到不同的调度结果. 表 3 给出了分别由 5 组节点优先权所生成的动态节点列表, 可以看出, 我们共得到从 (a) 到 (d)4 种不同的节点列表.

图 6(a) 给出了使用节点优先权 $\mathrm{bl}_{\mathrm{comp}}$ 得到的调度结果, 该结果使用了 3 个处理器, 调度长度为 18. 图 6(b) 给出了使用节点优先权 bl 得到的调度结果, 同样使用了 3 个处理器, 调度长度为 18. 图 6(c) 给出了使用 $\mathrm{bl}_{\mathrm{in}}$ 得到的调度结果, 其调度长度仍然是 18, 但是使用了 4 个处理器. 图 6(d) 给出了由 $\mathrm{bl}_{\mathrm{out}}$ 和 $\mathrm{bl}_{\mathrm{io}}$ 得到的相同节点列表的调度结果, 该结果使用 4 个处理器并且调度长度是 17, 优于前 3 个长度为 18 的结果. 以上高级动态列表调度方法得到的结果均优于经典方法, 并且有时还会减少所使用的处理器个数.

5.2 随机 DAG 比较

为了获得比某些特例更有说服力的结果, 随机图通常被用来比较不同的调度算法以获得统计性结果. 为此, 我们使用了一个基于 SDF^3 的随机 SDF 图生成器 [19], 并且在生成随机图时限定其为 DAG(无环图).

一个随机 DAG 可以从 5 方面约束: (1) 节点个数, (2) 平均入度, (3) 平均出度, (4) 节点的随机权重, (5) 边的随机权重. 本文假定平均入度和平均出度相同, 节点权重从 $w_{\min}$ 到 $w_{\max}$ 随机变化, 并使用通信运算比 (communication to computation ratio, CCR) 来生成边的随机权重. 这里把 CCR 定义

表 2　不同的静态节点列表及相应的关键子节点

Table 2　Different static node lists and corresponding critical children

Node priority	Static node list	Critical child								
		n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
$\mathrm{bl_{comp}}$ & $\mathrm{tl_{comp}}$	$n_1, n_4, n_3, n_2, n_8, n_7, n_6, n_5, n_9$	n_4	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl & tl	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
$\mathrm{bl_{in}}$ & $\mathrm{tl_{in}}$	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
$\mathrm{bl_{out}}$ & $\mathrm{tl_{out}}$	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
$\mathrm{bl_{io}}$ & $\mathrm{tl_{io}}$	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null

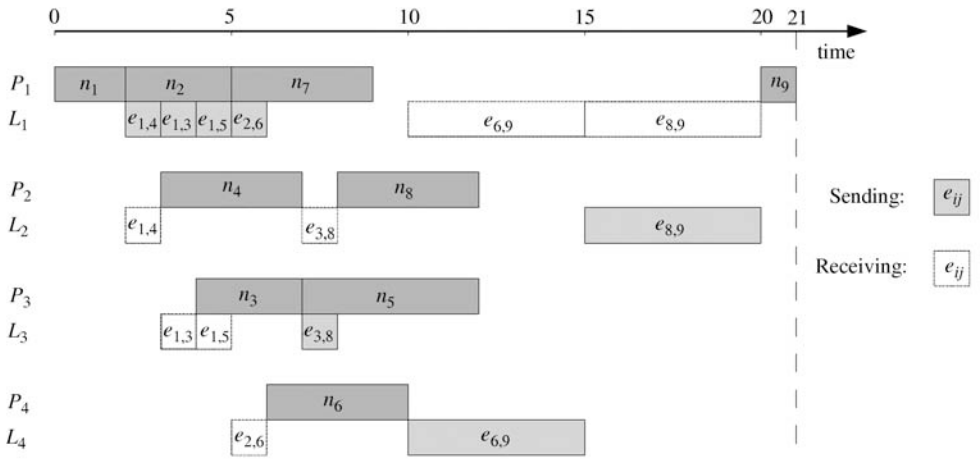

图 5　经典方法的调度结果

Figure 5　Schedule result of classic heuristic

表 3　不同的动态节点列表

Table 3　Different dynamic node lists

Node priority	Dynamic node list	No.
$\mathrm{bl_{comp}}$	$n_1, n_4, n_2, n_6, n_7, n_3, n_8, n_9, n_5$	(a)
bl	$n_1, n_4, n_2, n_7, n_6, n_3, n_8, n_9, n_5$	(b)
$\mathrm{bl_{in}}$	$n_1, n_2, n_4, n_3, n_8, n_6, n_7, n_9, n_5$	(c)
$\mathrm{bl_{out}}$	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$	(d)
$\mathrm{bl_{io}}$	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$	(d)

为边的平均权重与节点的平均权重之比, 即 CCR= $\frac{\frac{1}{|E|}\sum_{e\in E} c(e)}{\frac{1}{|V|}\sum_{n\in V} w(n)}$. CCR 的典型值 0.1, 1 和 10 分别表示低通信代价, 中等通信代价和高通信代价的情形. 边的权重则是从 $w_{\min}\times$ CCR 到 $w_{\max}\times$ CCR 随机变化.

高级动态列表调度方法可以使用 5 组节点优先权, 我们把这 5 组节点优先权和高级动态方法组合使用, 并选择具有最短调度长度的结果作为最终调度结果, 整个过程称为一个组合高级动态方法. 为了比较组合高级动态方法与使用 bl & tl 作为节点优先权的经典列表调度方法在性能上的区别, 我们

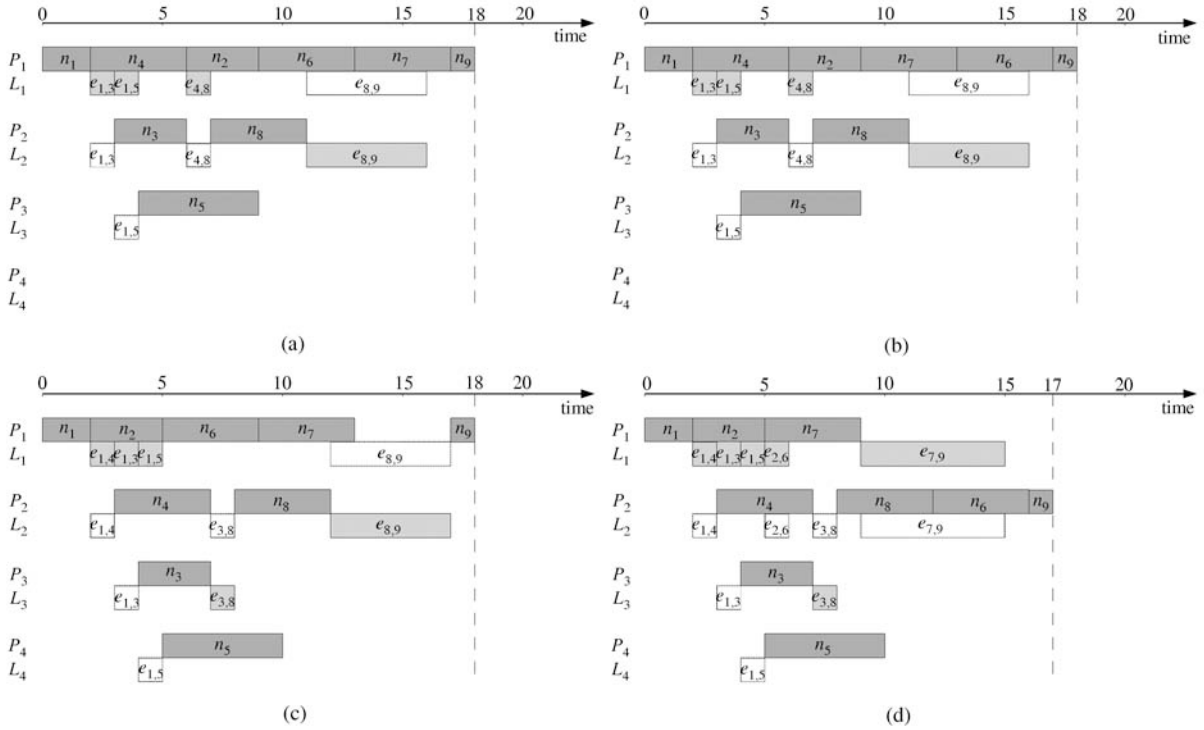

图 6 高级动态立标调度方法的调度结果

Figure 6 Schedule results of advanced dynamic heuristic

表 4 组合高级动态方法与经典列表调度方法的优劣对比

Table 4 Comparison of the combined advanced dynamic heuristic with the classic list scheduling heuristic

Average in/out degree	2			3			4		
CCR	0.1	1	10	0.1	1	10	0.1	1	10
Better	1.2%	86.4%	94.7%	1.9%	78.2%	95.6%	2.3%	76.6%	95.3%
Equal	24.2%	0.9%	0.0%	13.7%	0.0%	0.0%	8.7%	0.0%	0.0%
Worse	74.6%	12.7%	5.3%	84.4%	21.8%	4.4%	89.0%	23.4%	4.7%

使用以下随机 DAG: 固定节点数为 100, 节点权重从 $w_{\min} = 100$ 到 $w_{\max} = 1000$ 随机变化, 针对不同的平均出/入度和 CCR 生成 9 组随机 DAG, 每组包括 1000 个样本. 表 4 给出了组合高级动态方法与经典列表调度方法的优劣对比, 从中可以看出, 当 CCR=0.1 时组合高级动态方法在大多数情况下都不如经典方法, 但是随着 CCR 的增大, 组合高级方法在绝大多数情况下都会获得比经典方法更好的调度结果.

为了更加清楚地说明组合高级动态方法的性能, 定义加速因子为 $\mathrm{Acc} = \frac{sl_{\mathrm{classic}}}{sl_{\mathrm{advanced}}}$. 我们测试了 27 组随机 DAG, 图 7(a) 给出了组合高级动态列表调度方法的平均 Acc. 我们注意到在 CCR=1 和 CCR=10 的情况下, 使用组合高级动态列表调度方法可以使调度结果获得加速 (Acc>1), 同时平均 Acc 会随着 CCR 的增长而相应地提高, 当 CCR=10 时调度结果可以加速到高达 80%. 如果节点个数固定, 当 CCR=10 时平均 Acc 随着平均出/入度的增长而增长. 该现象可以解释为关键子节点技术在节点具有多个前趋时可以选择到更好的处理器, 出/入度越高, 关键子节点技术效果越好. 由于在数字

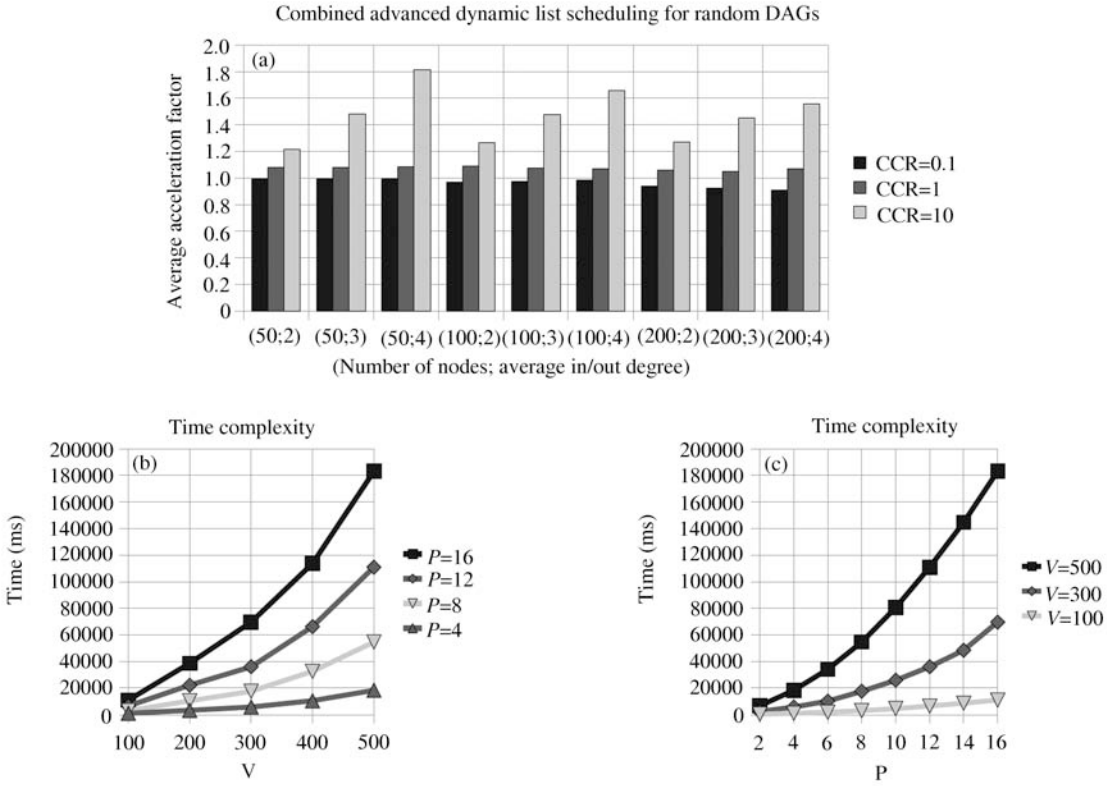

图 7　高级动态方法的平均加速因子 (Acc) 和时间复杂度

Figure 7　Average Acc of the advanced dynamic heuristic and its time complexity

(a) Average Acc; (b) time complexity with V; (c) time complexity with P

通信及视频压缩等现代嵌入式应用中的通信代价一直在增长, 本文的方法非常适合于在并行嵌入式系统中对这类应用进行任务调度.

5.3　高级动态方法的时间复杂度

经典列表调度方法的时间复杂度为 $O(PE^2O(\text{routing})+V^2)$, 其中 P, V 和 E 分别是处理器, 节点和边的个数. $O(\text{routing})$ 代表了链路上连接线的最大个数, 当并行嵌入式系统中使用静态路由策略时, 其通常是一个确定值. 关键子节点技术使时间复杂度提高了一个因子 P, 因此高级动态方法的时间复杂度为 $O(P(PE^2O(\text{routing})+V^2))$, 而它与 5 组节点优先权的组合并不增加时间复杂度的阶次.

图 7(b)(c) 给出了使用组合高级动态方法在不同处理器数目的系统结构上对各种规模的 DAG 进行调度所花费的时间. 这里所有 DAG 的平均出/入度均为 4, 各处理器分别通过不同的通信连接线连在同一个交换器上. 从图中可以看出, 所花费的时间随着 V 以及 P 的平方增长. 对于具有 500 个节点的 DAG 和 16 个处理器的系统结构, 我们在 2.4 GHz 的 Pentium 双核 PC 机上执行调度只需要大约 3 min. 事实上, 复杂的并行嵌入式应用在粗粒度或者中等粒度模型中的节点数通常不会超过 500, 而并行嵌入式系统中的 P 相比于 V 和 E 则更是小得多. 因此, 时间复杂度的增加对于快速原型设计方法来说是合理的并且是可以接受的.

6 结论

针对具有通信竞争的列表调度, 本文提出了 3 组新的节点等级 (top level 和 bottom level) 和 2 种高级技术 (关键子节点和通信延迟). 我们还给出了在异构并行嵌入式系统中使用这些新的节点等级和高级技术的动态列表调度方法. 新的节点等级考虑了通信竞争, 可以作为节点优先权来生成不同的节点列表; 关键子节点技术能为节点选择更好的处理器; 通信延迟技术在必要时通过延迟通信可以扩大连接线上的空闲时间区间.

对于给定的 DAG, 高级动态调度方法可以使用不同的节点列表从而得到不同的调度结果. 我们把 5 组节点节点优先权跟高级动态调度方法相结合并选取最佳结果, 整个过程称为组合高级动态列表调度方法. 为了同经典方法相比较, 我们在仿真实验中使用同构的并行嵌入式系统和随机生成的 DAG, 结果表明, 在中等通信代价和高通信代价的情况下, 组合高级动态方法可以有效地缩短调度长度. 本文方法可以在通信代价很高时使调度结果加速高达 80%, 在有些时候还可以减少硬件资源的使用. 由于数字通信及视频压缩等现代嵌入式应用中的通信代价正在从低到中甚至到高地增长, 本文方法将非常适合于在并行嵌入式系统中对这类应用进行任务调度.

致谢 感谢国家留学基金管理委员会提供资助, 感谢西安交通大学殷勤业教授对本文的撰写给出宝贵建议.

参考文献

1 Lee E, Parks T. Dataflow process networks. Proc IEEE, 1995, 83: 773–801
2 Sriram S, Bhattacharyya S S. Embedded Multiprocessors-Scheduling and Synchronization. New York, NY: Marcel Dekker Inc., 2000
3 Sarkar V. Partitioning and Scheduling Parallel Programs for Multiprocessors. Cambridge, MA: MIT Press, 1989
4 Garey M R, Johnson D S. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York, NY: W H Freeman & Co., 1990
5 Adam T L, Chandy K M, Dickson J R. A comparison of list schedules for parallel processing systems. Commun ACM, 1974, 17: 685–690
6 Kasahara H, Narita S. Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans Comput, 1984, 33: 1023–1029
7 Hwang J J, Chow Y C, Anger F D, et al. Scheduling precedence graphs in systems with interprocessor communication times. SIAM J Comput, 1989, 18: 244–257
8 Wu M Y, Gajski D. Hypertool: A programming aid for message-passing systems. IEEE Trans Parallel Distr Syst, 1990, 1: 330–343
9 Yang T, Gerasoulis A. DSC: Scheduling parallel tasks on an unbounded number of processors. IEEE Trans Parallel Distr Syst, 1994, 5: 951–967
10 Kwok Y K, Ahmad I. Dynamic critical-path scheduling: an effective technique for allocating task graphs onto multiprocessors. IEEE Trans Parallel Distr Syst, 1996, 7: 506–521
11 Sih G, Lee E. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans Parallel Distr Syst, 1993, 4: 175–187
12 Kwok Y K, Ahmad I. Bubble scheduling: a quasi dynamic algorithm for static allocation of tasks to parallel architectures. In: Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing. Washington, DC, 1995
13 Grandpierre T, Lavarenne C, Sorel Y. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In: Proceedings of 7th International Workshop on Hardware/Software Co-Design. Rome, 1999

14 Sinnen O, Sousa L. Communication contention in task scheduling. IEEE Trans Parallel Distr Syst, 2005, 16: 503–515
15 Tang X, Li K, Padua D. Communication contention in APN list scheduling algorithm. Sci China Ser F-Inf Sci, 2009, 52: 59–69
16 Sinnen O. Task Scheduling for Parallel Systems. Hoboken, NJ: John Wiley & Sons Inc., 2007
17 Kwok Y K, Ahmad I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surveys, 1999, 31: 406–471
18 Sinnen O, Sousa L. List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Comput, 2004, 30: 81–101
19 Stuijk S, Geilen M, Basten T. SDF^3: SDF for free. In: Proceedings of 6th International Conference on Application of Concurrency to System Design. Los Alamitos, 2006

Advanced list scheduling heuristic for task scheduling with communication contention for parallel embedded systems

MU PengCheng[1*], NEZAN Jean-François[2], RAULET Mickaël[2] & COUSIN Jean-Gabriel[2]

1 Ministry of Education Key Lab for Intelligent Networks and Network Security, School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049*, China;*
2 IETR/Image and Remote Sensing Group, CNRS UMR 6164/INSA Rennes, 35043 *Rennes Cedex, France*
*E-mail: pengchengmu@gmail.com

Abstract Modern embedded systems tend to use multiple cores or processors for processing parallel applications. This paper indeed aims at task scheduling with communication contention for parallel embedded systems and proposes three advanced techniques to improve the list scheduling heuristic. Five groups of node levels (two existing groups and three new groups) are firstly used as node priorities to generate node lists. Then the critical child technique improves the selection of a processor in the scheduling process. Finally, the communication delay technique enlarges the idle time intervals on communication links. We also propose an advanced dynamic list scheduling heuristic by combining the three techniques. Experimental results show that the combined advanced dynamic heuristic is efficient to shorten the schedule length for most of the randomly generated DAGs in the cases of medium and high communication. Our method accelerates an application up to 80% in the case of high communication and can also reduce the use of hardware resources.

Keywords list scheduling, communication contention, node level, critical child, communication delay