面向通讯同步的多处理器阵列重构

吴亚兰 武继刚 姜文超 刘竹松
(广东工业大学计算机学院 广州 510006)

摘 要 从多处理器阵列中获取所需大小并且同步通讯性能优良的子阵列,是高性能拓扑重构的核心问题之一。基于不同的逻辑列剔除策略提出了3种面向通讯同步的拓扑重构算法:基于分治思想剔除逻辑列的重构算法(SCA_01),该算法能够使被优化的逻辑列相对均匀地分布在物理阵列中;优先剔除长逻辑列的贪心重构算法(SCA_02),该算法能够使被优化的逻辑列的长链接总数最少;基于分治与长链接数的混成重构算法(SCA_03),该算法将某一区域内的最长逻辑列剔除,且尽可能将剩余逻辑列均匀分布在物理阵列中。同时,对逻辑阵列的最大通讯延时给出了下界的求解算法。实验结果表明,3种算法在故障率小于1%、逻辑列的剔除率超过20%时,算法重构出的逻辑阵列的通讯延时特别接近计算出的性能下界。在多数情况下SCA_01优于SCA_02和SCA_03,而后两者的性能相近。在小阵列上且故障率与剔除率较小时,SCA_02具有性能优势,但在大阵列上SCA_03具有优势。在32×32的阵列上,SCA_01构造的阵列产生的通讯延时较SCA_02和SCA_03产生的延时平均减少25%,并且运行速度也提升了19.4%。

关键词 VLSI阵列,拓扑重构,容错,分治,算法
中图法分类号 TP303 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.07.009

Reconfiguring Multiprocessor Arrays for Synchronous Communication

WU Ya-lan WU Ji-gang JIANG Wen-chao LIU Zhu-song
(School of Computer Science and Technology, Guangdong University of Technology, Guangzhou 510006, China)

Abstract Reconfiguring VLSI arrays to get a logical array with given size and synchronous communication is one of the key problems in reconfigurable topology for high performance computing. This paper presented three algorithms based on three different strategies of excluding logical co-lumns. The first algorithm, named SCA_01, can make the logical co-lumns in the uniform distribution in the host array, based on the divide-and-conquer for excluding logical columns. The second algorithm, named SCA_02, can minimize the number of the long interconnects of the logical array, based on the excluding the long logical column in priority. The third algorithm, named SCA_03, keeps the logical columns distributed in uniform way based on the hybrid strategy from excluding the long logical column and divide-and-conquer. In addition, this paper contributed an algorithm to calculate the lower bound of the communication delay for the given logical array. Simulation results show that, the communication delay of the logical array reconstructed by three algorithms is close to the lower bound proposed in this paper, when fault rate is less than 1% and the exclusion rate of logical columns is over 20%. Algorithm SCA_01 is superior to SCA_02 and SCA_03 in most cases, while SCA_02 and SCA_03 have nearly the same performance. But SCA_02 is better on smaller arrays and SCA_03 is better on large arrays, when the fault rate and exclusion rate are relatively small. The communication delay generated by SCA_01 is less than that of SCA_02 and SCA_03 by 25% on 32×32 host arrays. Moreover, SCA_01 is faster than the other two algorithms, and the running time is saved by 19.4%. It is concluded that SCA_01 is one of the relatively desirable algorithms to generate the logical arrays with minimum communication delay for high performance computing.

Keywords VLSI array, Topology reconstruction, Fault-tolerance, Divide and conquer, Algorithm

1 引言

随着超大规模集成电路(Very Large Scale Integrated, VLSI)和晶片规模集成技术(Wafer Scale Integration, WSI)的不断发展,单一芯片上能集成更多的处理器单元(Processing Elements, PEs),但同时在制造和运行过程中PEs发生故障的可能性也随之增加,进而影响系统的稳定性和可靠性。因此,针对含有故障PEs的VLSI阵列,需要提出有效的容错技

到稿日期:2016-08-20 返修日期:2016-11-03 本文受国家自然科学基金项目(61572144),广东省科技计划应用专项基金(2015B010129014),广东省自然科学基金项目(2016A030313703)资助。

吴亚兰(1993-),女,硕士生,主要研究方向为高性能计算,E-mail:wuyalan93@126.com;武继刚(1963-),男,博士,教授,主要研究方向为高性能计算;姜文超(1977-),男,博士,讲师,主要研究方向为云计算、高性能计算、分布式系统等;刘竹松(1979-),男,博士,副教授,主要研究方向为云计算安全和大数据。

术，以提高系统的稳定性和可靠性。

重构处理器阵列的常用方法有冗余方法（Redundancy Approach）和降阶方法（Degradation Approach）。冗余方法指在生产制造时加入备用 PEs，用其代替那些发生故障的 PEs 以实现阵列的重构[1-3]。阵列冗余重构技术在实际中有许多重要的应用，如监控器阵列的自修复[4]、运动检测系统[5]、现场可编程门阵列（Field Programmable Gate Array，FPGA）的动态恢复[6]；其缺点是处理器阵列的维数是固定的，若备用处理器数量太少，则无法完全代替所有发生故障的处理器，系统将停止运行，导致该方法失效。降阶方法不需要加入备用 PEs，当阵列中出现故障的 PEs 时，就在原阵列中通过算法技术，利用剩余的无故障 PEs 构造新的逻辑阵列。因为重构后的阵列的阶数比原始阵列的阶数小，所以该方法称为降阶方法。

文献[1]总结了 1990 年之前具有显著意义的降阶方法。文献[7]研究了在行/列穿越、行穿越/列选路以及行/列选路 3 种不同的选路策略下的重构问题，并且证明了在不同的选路策略下大多数的重构问题是 NP 难解问题。

对于可重构处理器阵列的上界（最大可用处理器阵列的大小）问题，文献[8]通过一种无限制的补偿方式得到其上界，但这种方式得到的上界过大，在故障 PE 非均匀分布的情况下尤为明显。文献[9]提出了更为准确的计算上界的算法，尤其针对故障 PE 聚集型的阵列，其极大地降低了文献[8]获得的上界值。文献[10]阐明了影响逻辑列总数的瓶颈条件，并突破限制逻辑列增长的瓶颈，提出新的求解算法来获得更为准确的新上界。

文献[8,11]提出了基于贪心思想的算法来构造最大的目标阵列（Maximum Target Array，MTA）。在多处理器系统实时重构时，需要减少重构算法的执行时间。目前出现了许多加速重构的算法，如部分重选路技术[12-13]、简化行选路方案[14]、启发式算法[15]和遗传算法[16-17]、基于整合行和列重选路策略[18]、使用预处理的方式[19-20]。由于阵列的内部链接长度减小，阵列的通讯延时、路径花费和动态功耗也会随之减小，因此文献[21]提出了一种基于动态规划的算法，用来优化文献[8]构造的 MTA。文献[22]提出了一种新颖的重构技术来构造一个紧密耦合目标阵列，且确定了 MTA 的总互连长度的下界。在处理器阵列的实际运行中，处理器的功耗是不可回避的问题。在阵列处理器温度已知的情况下，文献[23]使用动态规划技术构造低温目标阵列，以保证每条逻辑列的整体温度在给定区域内达到局部最优。在研究处理器重构算法时，研究者通常假定其开关是无故障的，而文献[24]提出并讨论了开关容错的多处理器重构算法，通过对开关故障所在区域的处理器进行预处理，完成逻辑列的重构。三维集成技术是当前多处理器系统的发展趋势，文献[25]系统地研究了三维阵列的容错重构技术，提出的启发式算法能在线性时间内构造最大的逻辑阵列。

在阵列的高性能计算应用方面，PEs 间通讯的同步问题显得尤为重要，我们在前期[26]已提出了目标阵列同步通讯的优化问题，并对给定的逻辑阵列实现了相应的性能优化，但对于如何构造用户所需大小的目标阵列及对其实现同步通讯性能的优化，目前还没有相关的研究。

本文面向高性能计算中阵列处理器结构的特点，针对如何构造用户指定大小并且同步通讯性能优良的逻辑子阵列进行拓扑重构的探索性研究。由于具有同步通讯性能要求的逻辑子阵列的构造问题明显难于具有 NP 难解的普通逻辑子阵列的构造问题，本文围绕逻辑列的筛选策略，对构造用户指定大小并且同步通讯性能优良的逻辑子阵列首次展开启发式算法研究，主要贡献如下。

（1）提出了 3 种不同的逻辑列的筛选策略，并由此分别导出 3 种不同的面向同步通讯的逻辑子阵列重构算法。3 种算法的基本思想分别为：基于分治思想的逻辑列剔除、基于链接长度的逻辑列剔除以及基于分治与链接长度混成的逻辑列剔除。3 种算法在不同的情况下各具优势，但在多数情况下，基于分治思想的逻辑列剔除策略更能体现出优化效果。

（2）对于给定的逻辑阵列，本文给出了计算最大通讯延时下界的算法，并且实验结果证明了在故障率较小、逻辑列的剔除率较大时，提出的 3 种算法的效果均十分接近下界。

（3）在不同的物理阵列上进行了模拟实验，对比了提出的 3 种算法的优化性能，获得了每种算法的优势区间，为面向同步通讯的子阵列的构造奠定了算法研究的基础。

2 基本概念与相关算法

2.1 基本概念

在本文中，将出厂后直接获得的原始处理器阵列称为主阵列（或物理阵列），记为 H。在阵列 H 中随机分布着一些有故障的 PEs。由阵列 H 中无故障的 PEs 重构得到的子阵列称为逻辑阵列（目标阵列），记为 $L(T)$。假设阵列 H 的故障率为 $\rho(0<\rho<1)$，表示在 $m\times n$ 的主阵列中有 $\rho\cdot m\cdot n$ 个有故障的 PEs。主阵列 H 中的行（列）称为物理行（列），逻辑阵列 L 中的行（列）称为逻辑行（列）。

物理阵列中所有相邻的 PEs 都是通过一个 4 个端口的开关相互链接的，且阵列中所有的开关和链接都被假定为无故障的[21,26]。图 1 展示了一个 4×4 的主阵列的体系结构、开关状态和选路策略。图中的每个正方形表示一个 PE，每个小圆圈表示一个开关。灰色的正方形表示该 PE 是有故障或不可用的，而白色的正方形代表该 PE 是无故障或可用的。

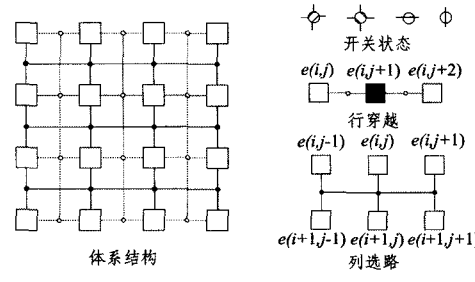


图 1 一个 4×4 主阵列的链接方式与开关状态

主阵列 H 中第 i 行、第 j 列的 PE 用 $e(i,j)$ 来表示。如图 1 所示，有 4 种开关状态，PEs 可以通过改变周围开关的状态来链接上、下、左、右 4 个方向上的 8 个 PE。假设 $e(i,j+1)$ 是有故障的，如果 $e(i,j)$ 可以直接绕过 $e(i,j+1)$ 与 $e(i,j+2)$ 通信，则将该策略称为行绕行策略。同理，列绕行策略也可如

此定义。如果 $e(i,j)$ 可通过外部开关与 $e(i+1,j')$ 通信，且 $|j'-j|\leqslant d$，则称这样的策略为列重选路策略，其中，d 称为补偿距离且设定为 1[8]。同理，行选路策略也可类似定义。本文采用的选路策略是行/列重选路策略。

一个逻辑阵列中有 6 种可能的链接方式[21]。基于开关的使用，可分为两种类型：

1)短链接，即使用一个开关来链接相邻的 PEs；

2)长链接，即使用两个开关来链接相邻的 PEs。

如图 2 所示，图 2(a)和图 2(d)属于短链接，图 2(b)、图 2(c)、图 2(e)和图 2(f)属于长链接。图 2(a)－图 2(c)适用于行选路的情况，而图 2(d)－图 2(f)适用于列选路情况。不难理解，长链接导致 PEs 间的通讯存在额外延时。

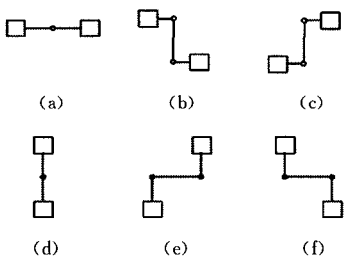


图 2　长链接与短链接

2.2　相关算法

与本文相关的算法主要有如下 3 个，即贪心列选路算法(Greedy Column Rerouting，GCR)[8]、基于动态规划的局部最优逻辑列优化算法(Local optimal column by Dynamic Programming，LDP)[21]和阵列的同步性能优化算法(Synchronization Performance Optimization，SPO)[26]。

在行/列重构的约束下，GCR 算法是一种基于贪心思想的启发式算法，利用它来构造一个最大的逻辑阵列 MTA。GCR 算法以从左到右的方式构造目标阵列。假设 MTA 有 k 条逻辑列，该算法首先在被选择的行中构建第一条最左逻辑列，然后依次在 $i-1$ 条最左逻辑列的右边区域构造第 i 条最左逻辑列，其中 $i=2,3,\cdots,k$。关于 GCR 算法更详尽的描述可参考文献[8]。

假设 B_l 和 B_r 是阵列中的两条逻辑列，若 B_l 中每个 PE 都处于 B_r 中的每个 PE 的左边，则称 $B_l<B_r$。本文中用 $A[B_l, B_r]$ 表示边界为 B_l 和 B_r 的区域所包含的 PEs 区域(包含边界 B_l 和 B_r)；同理，用 $A[B_l, B_r)$ 表示边界为 B_l 和 B_r 的区域所包含的 PEs 区域(包含边界 B_l，但不包含边界 B_r)。逻辑列 B_l 和 B_r 分别被称为左边界和右边界。

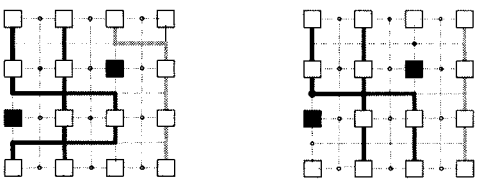
LDP 算法是阵列从右到左对逻辑列进行优化的算法。假设通过 GCR 算法构造的 MTA 从左到右的逻辑列分别记为 $C_1', C_2', \cdots, C_k'$。LDP 算法首先在 C_k' 的右边区域重构第 k 条逻辑列，选取该区域中构造的“最直”且最右的一条逻辑列，记为 C_k；然后在区域 $A[C_{k-1}', C_k)$ 重构一条区域中“最直”且最右的逻辑列，记为 C_{k-1}，以此类推，在区域 $A[C_{k-2}', C_{k-1})$ 中重构逻辑列，记为 $C_{k-2}, \cdots$，直到在区域 $A[C_1', C_2)$ 中构造出 C_1；最终构造的逻辑列 $C_i(1\leqslant i\leqslant k)$ 组成该算法的目标阵列，详见文献[21]。

SPO 算法旨在使阵列在通讯上尽可能地达到同步，使阵列工作时产生的通讯延时尽可能少。在 SPO 算法中，通过调整阵列长链接的位置，使长链接的分布尽可能地集中在某些物理行。假设要同步的阵列大小为 $m\times n$，该算法首先扫描整个阵列，确定每条长链接可移动的范围，并将其分别存入对应的集合中；接着对所有集合进行求交运算，使得尽可能多的集合合并到尽可能少的交集中，且将参与求交的集合替换为交集；最后根据集合求交运算的结果来移动长链接。更多细节，请参考文献[26]。

3　面向通讯同步的重构算法

在逻辑阵列中，若相邻的两行处理器单元之间存在一个或者一个以上的长链接，则称这两行之间存在一个单位的额外通讯延时[16]；反之，则表示这两行之间不存在额外通讯延时。我们定义整个阵列产生的通讯延时是各行额外延时的总和。

使用图 3 来阐明本文的研究动机。图 3(a)与图 3(b)均为 4×3 的逻辑子阵列，但图 3(a)的额外通讯延时为 3 个单位，分别位于第一行与第二行、第二行与第三行、第三行与第四行之间；而图 3(b)仅为 1 个单位，位于第二行与第三行之间。在高性能计算、图形图像处理等应用中，PEs 间通讯的同步将直接影响算法的性能。从体系结构的角度，能够为用户提供具有良好同步通讯性能的子阵列是十分有意义的。因此图 3(b)是我们推崇的逻辑阵列，构造这样的逻辑阵列也是本文的目标所在。



(a)普通阵列　　(b)同步通讯优化的逻辑阵列

图 3　普通逻辑阵列与同步通讯优化的逻辑阵列

本文求解的问题可描述如下。

问题 P_c：给定一个 $m\times n$ 的容错可重构主阵列 H 以及正整数 r,s，其中 $r\leqslant m, s\leqslant n$，在行列均可选路的方式内，构造一个大小为 $r\times s$ 并且具有最少额外通讯延时的逻辑子阵列 T。

关于该问题的难度，给出如下说明。

问题 P：给定一个 $m\times n$ 的容错可重构主阵列 H 与正整数 r,s，其中 $r\leqslant m, s\leqslant n$，在行列均可选路的方式内，构造一个大小为 $r\times s$ 的逻辑子阵列 T。

文献[7]针对问题 P 的难度给出了论证，指出问题 P 是 NP 难解的。不难理解，解算问题 P_c 的任意一个算法都能解算问题 P，因此问题 P_c 的难度大于问题 P，从而问题 P_c 也是 NP 难解的。

本节将对问题 P_c 给出 3 个逻辑列剔除的策略，并由此产生 3 个启发式算法，用来构造同步通讯性能优良的逻辑阵列。

3.1　主算法框架

本文提出了 3 种逻辑列的剔除策略，分别用 Extract_01，Extract_02，Extract_03 表示。基于 3 种策略的重构算法的框架是一致的，算法的基本步骤为：1)使用剔除策略对逻辑阵列

的逻辑列进行剔除，剩余的逻辑列即为目标阵列的初始阵列；2)将通过剔除而生成的初始阵列进行 LDP 算法优化；3)将优化后的阵列使用 SPO 算法进行同步性能的提升，得到问题 P_c 的近似解，并将其作为目标阵列。由于不同的剔除策略导致主算法产生不同的目标阵列，因此本文将这些剔除策略分别对应如下 3 种重构算法，分别记作 SCA_01，SCA_02，SCA_03。3 种算法的形式化描述如下。

算法 1 SCA_0*i*

```
/*主算法框架，其中 i:=1,2,3*/
输入：由 GCR 算法构造且经 LDP 算法优化的逻辑阵列 L；
输出：目标阵列 T
Begin
  L′:=Extract_0i (L);/*使用第 i 个策略剔除逻辑列*/
  L′:=LDP(L′);/*使用动态规划算法减少长链接数*/
  T:=SPO(L′); /*同步优化*/
End.
```

3.2 逻辑列的剔除策略与算法

3.2.1 基于分治思想的均匀剔除

子算法 Extract_01 基于分治策略的思想，递归地剔除各个区域处于中间位置的逻辑列，使得剩余的逻辑列尽可能均匀地分布在物理阵列中。子算法 Extract_01 先将整个阵列作为一个区域，选取处于区域中间位置的逻辑列进行剔除，若逻辑列的个数为偶数，则选择剔除左边的逻辑列。以剔除的逻辑列为分界线将该区域分为两个子区域，然后分治剔除每个区域中处于中间位置的逻辑列并以剔除的逻辑列为分界线对区域进行划分，直至所剩逻辑列数满足要求(逻辑列数等于 s)时为止。

在如下的算法描述中，用 $L(l\cdot\cdot r)$表示逻辑阵列 L 中由第 l 条逻辑列至第 r 条逻辑列组成的子阵列。用 i 记录阵列中被剔除的逻辑列数量。

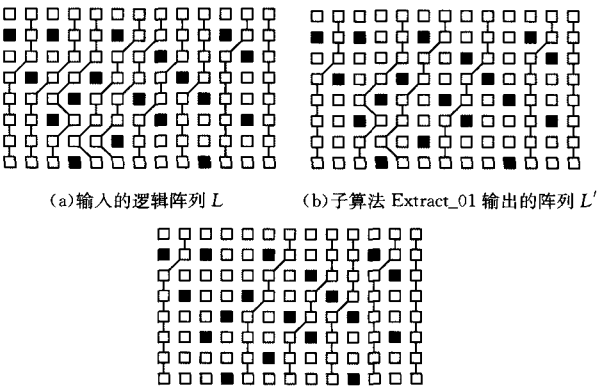
算法 2 Extract_01(i,$L(l\cdot\cdot r)$)

```
/*基于分治思想的均匀剔除*/
输入：m×k 逻辑阵列 L
输出：目标阵列的初始阵列 L′(m×s)
Begin
if i≤k−s then
  begin
    d:=⌊(l+r)/2⌋
    剔除第 d 条逻辑列；
    i:=i+1;
    Extract_01(i,L(l··d−1));
    Extract_01(i,L(d+1··r));
  end;
End.
```

为了更好地理解算法，下面举例说明算法的执行过程。如图 4(a)所示，H 是 8×13 的物理阵列，算法输入的阵列大小为 8×9，将从左至右的逻辑列记为 C_1'，C_2'，C_3'，C_4'，C_5'，C_6'，C_7'，C_8'，C_9'。假设要在该逻辑阵列上剔除 3 条逻辑列，具体步骤如下。

首先将编号位于中间的逻辑列 C_5'剔除。由于仍然需要继续执行剔除操作，算法在 C_1'，C_2'，C_3'，C_4'上剔除一条逻辑列 C_2'，同理在 C_6'，C_7'，C_8'，C_9'中剔除一条逻辑列 C_7'。目前已经剔除 3 条逻辑列，至此算法结束。效果如图 4(b)所示。

在主算法 SCA_01 中，先对阵列 L'使用 LDP 算法进行优化，随后使用 SPO 算法进行同步性能提升，最后获得目标阵列 T，如图 4(c)所示。



(a)输入的逻辑阵列 L (b)子算法 Extract_01 输出的阵列 L'

(c)经 LDP 和同步性能优化算法优化后的目标阵列 T

图 4 算法 SCA_01 示例

依据算法描述，最坏情况下要完成 $k-s$ 条逻辑列的剔除工作，这种剔除工作仅仅使用集合元素的删除操作即可实现，其递归深度为 $O(k-s)$，故算法的时间复杂度与空间复杂度均为 $O(k-s)$。

3.2.2 基于长链接数的贪心剔除

子算法 Extract_02 基于贪心策略的思想，首先统计输入的阵列中的每条逻辑列的长链接数，并依次剔除长链接数最多且最靠近左边的逻辑列，直至所剩逻辑列数满足要求(逻辑列数等于 s)时为止。

在算法描述中，用 $L(l\cdot\cdot k)$表示逻辑阵列 L 中由第 l 条逻辑列至第 k 条逻辑列组成的子阵列。

算法 3 Extract_02($L(1\cdot\cdot k)$)

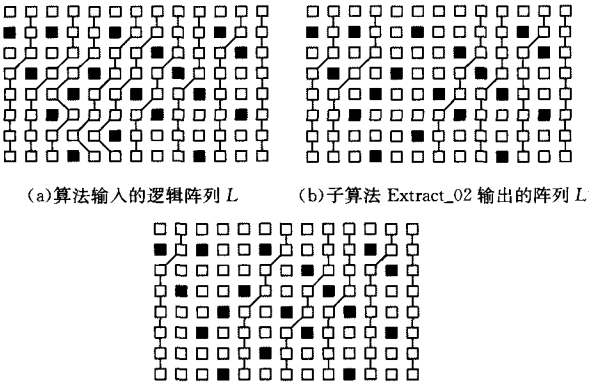
```
/*基于长链接数的贪心剔除*/
输入：m×k 逻辑阵列 L;
输出：目标阵列的初始阵列 L′(m×s);
Begin
1. 在逻辑列 l1,l2,…,lk 中选择前 k−s 条较长的逻辑列，记作 li1,
   li2,…,lik−s;
2. 将 L 中的 li1,li2,…,lik−s 剔除;
End.
```

为了更好地展示算法，下面举例说明算法的执行过程。使用如图 5(a) 所示的阵列，H 是 8×13 的物理阵列，算法输入的阵列大小为 8×9。其逻辑列从左至右依次记为 C_1，C_2，C_3，C_4，C_5，C_6，C_7，C_8，C_9。假设要在该逻辑阵列上剔除 3 条逻辑列，具体步骤如下。

首先，由图 5(a)可知长链接数最多的逻辑列有 3 条，当长链接数相等时，优先剔除左边的逻辑列，即剔除逻辑列 C_3；由于还需要剔除逻辑列，算法依次剔除逻辑列 C_4 与 C_5。目前已经剔除 3 条逻辑列，至此算法结束，如图 5(b)所示。

在主算法 SCA_02 中，先对阵列 L' 使用 LDP 算法进行

优化，随后再使用 SPO 算法进行同步性能提升，最后获得目标阵列 T，如图 5(c)所示。



(a)算法输入的逻辑阵列 L　(b)子算法 Extract_02 输出的阵列 L'

(c)经 LDP 和 SPO 算法优化后的目标阵列 T

图 5　算法 SCA_02 示例

该算法的第一步可用顺序统计算法[27]在 $O(k)$时间内完成；第二步为标记工作，可在 O(1)时间内完成。因此该算法的复杂度为 $O(k)$。

3.2.3　基于分治与长链接数的混成剔除

子算法 Extract_03 是 Extract_01 和 Extract_02 两个子算法的结合。子算法 Extract_03 首先将整个阵列视为一个区域，将该区域中长链接数最多的逻辑列剔除。若长链接数最多的逻辑列不止一条，那么选择剔除最靠近中间位置的逻辑列。以选中剔除的逻辑列为分界线，将该区域分为两个子区域，然后分治剔除每个子区域中长链接数最多的列并进行区域划分，直至所剩逻辑列数满足要求(逻辑列数等于 s)时为止。

在下面的算法描述中，用 $L(l\cdot\cdot r)$表示逻辑阵列 L 中由第 l 条逻辑列至第 r 条逻辑列组成的子阵列。用 i 记录阵列中被剔除的逻辑列数量。

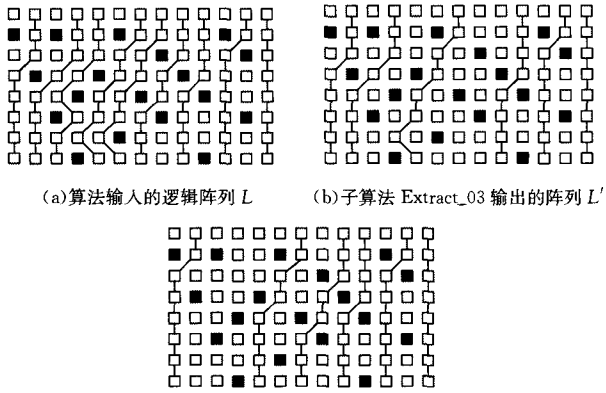
算法 4　Extract_03(i,$L(l\cdot\cdot r)$)

```
/*基于分治与长链接的混成剔除*/
输入：m×k 逻辑阵列 L
输出：目标阵列的初始阵列 L'(m×s)
Begin
if i≤k−s then
  begin
    d:=最靠近中间的最长逻辑列编号;
    剔除第 d 条逻辑列;
    i:=i+1;
    Extract_01(i,L(l..d−1));
    Extract_01(i,L(d+1..r));
  end;
End.
```

为了更好地展示算法思想，下面用示例来说明算法的执行过程。如图 6(a)所示，H 是 8×13 的物理阵列，算法输入的阵列大小为 8×9。其逻辑列从左至右依次表示为 C_1，C_2，C_3，C_4，C_5，C_6，C_7，C_8，C_9。假设要在该逻辑列上剔除 3 条逻辑列，具体操作步骤如下。

首先将最靠近中间位置的最长逻辑列 C_5 剔除，由于仍需要进行剔除操作，算法在 C_1，C_2，C_3，C_4 上剔除一条逻辑列 C_3，同理在 C_6，C_7，C_8，C_9 中剔除一条逻辑列 C_6。至此，已剔除 3 条逻辑列，算法结束。输出阵列如图 6(b)所示。

在主算法 SCA_03 中，先对阵列 L'使用 LDP 算法进行优化，随后再使用 SPO 算法进行同步性能提升，最后获得目标阵列 T，如图 6(c)所示。



(a)算法输入的逻辑阵列 L　(b)子算法 Extract_03 输出的阵列 L'

(c)经 LDP 算法和 SPO 算法优化后的目标阵列 T

图 6　算法 SCA_03 示例

依据算法描述，最坏情况下要完成 $k-s$ 条逻辑列的剔除工作，递归深度为 $O(k-s)$。故算法的复杂度为 $O(k-s)$。

依据主算法描述，SCA_01，SCA_02 和 SCA_03 算法第二步和第三步的复杂度均为 $O(m\cdot s)$，3 种算法的复杂度的区别仅在于其子算法，而子算法的算法复杂度对主算法的复杂度并无影响，所以 3 种算法的复杂度均为 $O(m\cdot s)$。

4　通讯延时的下界与算法说明

对于给定的逻辑阵列 $L=\{l_1,l_2,\cdots,l_s\}$，本节给出计算其通讯延时下界的算法。计算所得的通讯延时的下界可以直接用来评估任何同步通讯优化算法的性能。

令 *lowb* 表示 L 通讯延时的下界，l_i 表示最长的逻辑列。*lowb* 的计算起始于最长逻辑列的长链接数，即 *lowb* 的初始值设定为这条 l_i 的长链接数。然后逐次计算两条逻辑列 l_i 与 $l_j(j\neq i)$构成的子阵列$\{l_i,l_j\}$的最大通讯延时，而此时其他逻辑列的通讯延时设定为 0。检测 l_j 的每一条长链接的可移动范围是否被 l_i 的长链接的移动范围涵盖。若没有被涵盖，则子阵列$\{l_i,l_j\}$的最大通讯延时在 *lowb* 的基础上不得不增加 1 个单位，即使 l_j 的长链接可以与其他逻辑列的长链接保持同步。这一检测过程对每一个子阵列$\{l_i,l_j\}(j=1,2,\cdots,s,j\neq i)$都要计算一次，累计 *lowb* 的值，并更新长链接的可移动范围。计算 *lowb* 的具体算法可描述如下。在如下的算法描述中，用集合 *set* 和 *set*1 记录长链接的可移动范围。

算法 5　LowBound(L)

```
/*阵列通讯延时下界的计算*/
输入：逻辑阵列 L={l1,l2,…,ls}
输出：通讯延时下界 Lowb
Begin
  lowb:=li 的长链接数;
  set:=li 的所有长链接可移动范围;
  while (j≤s) and (j≠i) do
```

```
begin
  for lj 中的每一条长链接 do
    set1:=该长链接的可移动范围;
    if set1∩set=∅ then
      begin
        lowb:=lowb+1;
        set:=set1∪set;
    end;
  endfor;
end;
End.
```

对 SPO 算法中两种特殊情况的处理说明如下。

1)当剔除逻辑列后,在 LDP 算法进行优化时阵列的某个部分产生了如图 7(a)所示的情况,其中逻辑列 C_1''和 C_2''未优化,而 C_3' 是 LDP 算法优化后的一条逻辑列。经 LDP 算法优化后,该部分变成图 7(b)所示的情况。在执行同步性能优化算法时,逻辑列 C_2' 的两条长链接对应的可移动范围的集合必然会合并到同一个集合中,所以两条长链接要移动的位置也相同。当两条长链接移动到同一位置时,因两条长链接方向不同,导致两条长链接会互相"抵消",如图 7(c)所示。在这种特殊情况下,同步性能优化算法会减少同步阵列的长链接数。

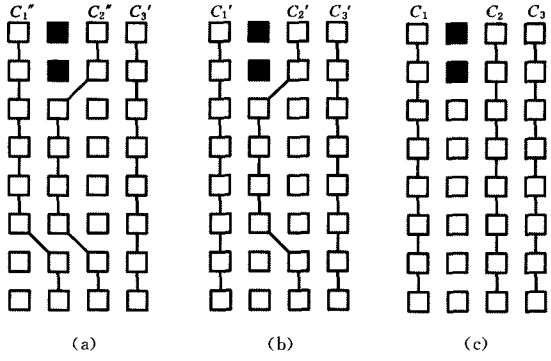


图 7 特殊情况(1)

2)阵列中某条逻辑列的长链接出现了如图 8 所示的 4 种情况。在对阵列进行同步时,为防止长链接错位而导致结果错误,对其长链接的可移动范围进行限制。对于图 8(a)和图 8(b)的情况,限制长链接 1 的可移动范围只能在其上方,长链接 2 的可移动范围只能在其下方。而对于图 8(c)和图 8(d)的情况,长链接 3 的可移动范围只能在其上方,长链接 5 的可移动范围只能在其下方,而长链接 4 不可移动。同理,若存在类似于如图 8 所示的情况且长链接数大于 3,则最上面那条长链接的可移动范围只能处于其上方,最下面那条长链接的可移动范围只能在其下方,处于两者中间的所有长链接均不可移动。

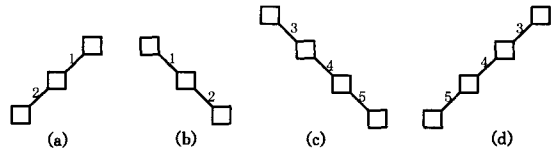


图 8 特殊情况(2)

5 实验结果与分析

实验构造的阵列沿用了以往的假设与模型[21,26],主阵列中有故障的 PEs 是随机定位的。对于故障率为 1%,5%,10%,15%的 32×32 和 128×128 阵列,分别构造了 10 个随机的物理阵列。对每个阵列在剔除率为 1%,5%,10%,15%,20%,25%,30%,35%,40%的情况进行实验数据收集,实验结果为每种情况下 10 个随机阵列经过不同的算法处理后取得的数据平均值。3 种算法均采用 C++语言实现,其实验结果是在 Intel(R) Core(TM) i5-3210M CPU 2.50GHz 8GB RAM 的机器上运行得出的。

在阵列大小为 32×32 且故障率不同的情况下,对 3 种算法的目标阵列产生的通讯延时及其同步前阵列的通讯延时的下界做了测试,结果如图 9—图 12 所示。图中的实际值表示算法导出的逻辑阵列的通讯延时,下界表示计算的通讯延时的下界。图 9(a)—图 9(c)分别展示了在故障率为 1%且剔除率不同时,算法 SCA_01,SCA_02 和 SCA_03 的目标阵列实际产生的通讯延时和在同步之前阵列产生的通讯延时的下界。当剔除率小于 30%时,算法 SCA_01 的目标阵列产生的通讯延时比算法 SCA_02 和算法 SCA_03 的目标阵列产生的通讯延时更接近于其通讯延时的下界;当剔除率大于 30%时,SCA_02 的目标阵列产生的通讯延时比算法 SCA_01 和算法 SCA_03 的目标阵列产生的通讯延时更接近其通讯延时的下界。由图 9—图 12可知,当故障率大于 1%时,算法 SCA_01 的目标阵列的通讯延时比算法 SCA_02 和算法 SCA_03 的通讯延时更接近于其通讯延时的下界。算法 SCA_02 和算法 SCA_03所构造的阵列产生的通讯延时与通讯延时的下界的接近程度大致相同。在故障率大于 1%时,算法 SCA_01 的性能比算法 SCA_02 和算法 SCA_03 的性能更好,而算法 SCA_02 和算法 SCA_03 的性能较为接近,所以此时算法 SCA_01 是较可取的重构算法。从纵向来看,随着故障率的减小,3 种算法所构造的阵列产生的延时越来越接近于下界。在故障率一定时,随着剔除率的增大,3 种算法产生的阵列的延时就越接近于下界;甚至在故障率小于 1%且剔除率较大时,算法构造出的阵列所产生的延时达到了下界值。

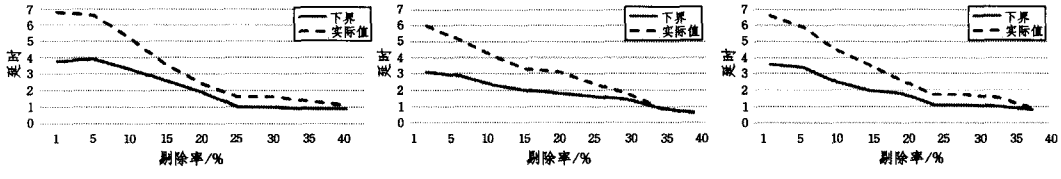
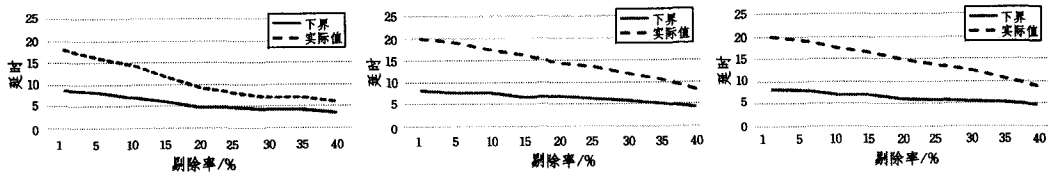
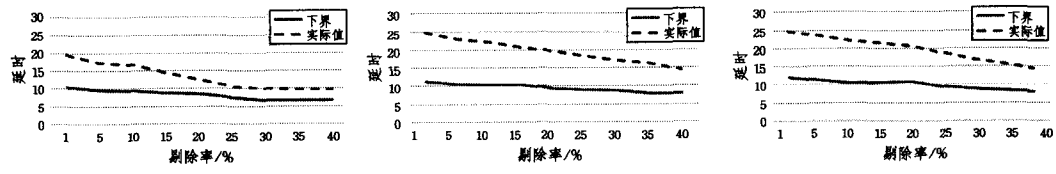


图 9 3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=1%,阵列大小为 32×32)



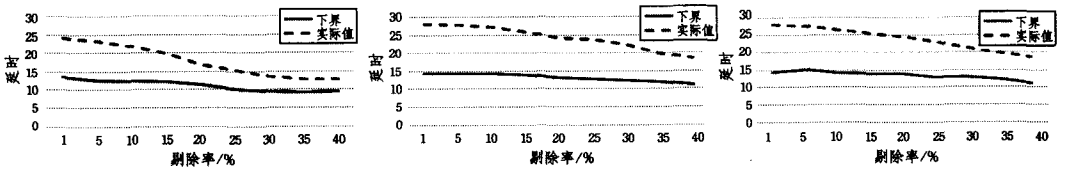
(a)SCA_01　　(b)SCA_02　　(c)SCA_03

图 10　3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=5%，阵列大小为 32×32)



(a)SCA_01　　(b)SCA_02　　(c)SCA_03

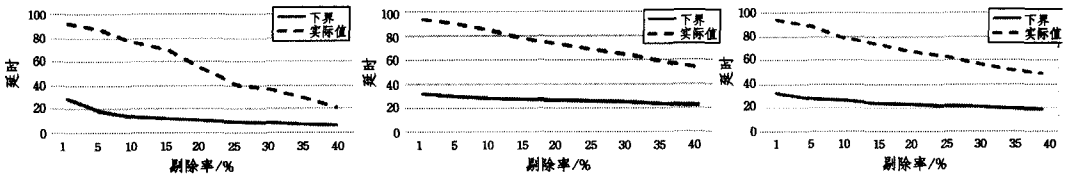
图 11　3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=10%，阵列大小为 32×32)



(a)SCA_01　　(b)SCA_02　　(c)SCA_03

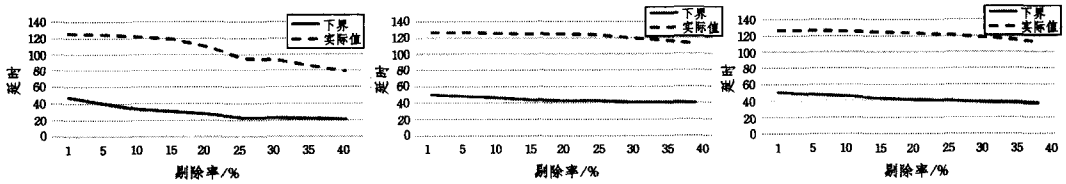
图 12　3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=15%，阵列大小为 32×32)

对大小为 128×128 的阵列进行了同样的实验测试，结果如图 13－图 16 所示。同理，图中的实际值表示算法导出的逻辑阵列的通讯延时，下界表示计算的通讯延时的下界。由图 13 可知，算法 SCA_01 的目标阵列产生的通讯延时要比算法 SCA_02 和算法 SCA_03 的目标阵列产生的通讯延时更接近其通讯延时的下界。由图 14－图 16 可知，当故障率大于 1%时，3 种算法的目标阵列产生的通讯延时远离其下界。因为在计算阵列的通讯延时下界时，统计长链接的可移动范围忽略了阵列中其他逻辑列的存在。当阵列的故障率较大或者剔除率较小时，逻辑列之间的关联性较大，而在计算通讯延时下界时忽略了阵列中逻辑列之间的相关性，导致其通讯延时下界与同步优化后的阵列产生的通讯延时相差较大。



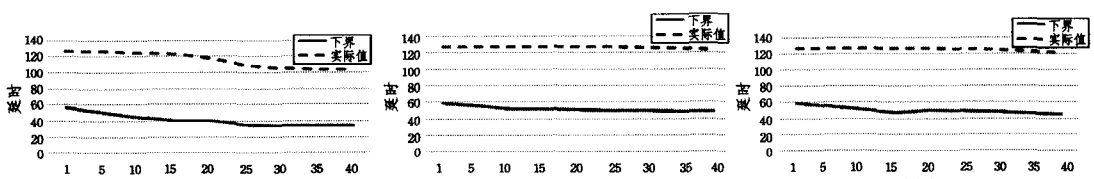
(a)SCA_01　　(b)SCA_02　　(c)SCA_03

图 13　3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=1%，阵列大小为 128×128)



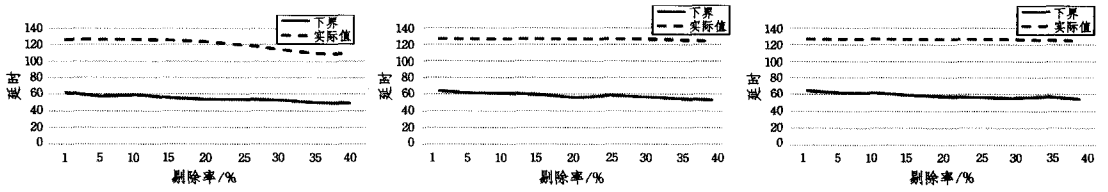
(a)SCA_01　　(b)SCA_02　　(c)SCA_03

图 14　3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=5%，阵列大小为 128×128)



(a)SCA_01　　(b)SCA_02　　(c)SCA_03

图 15　3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=10%，阵列大小为 128×128)



(a)SCA_01 (b)SCA_02 (c)SCA_03

图 16 3 种算法生成的逻辑阵列的通讯延时与下界的对比(ρ=15%,阵列大小为 128×128)

在 32×32 阵列的不同故障率下,对 3 种算法的目标阵列产生的通讯延时进行了对比测试,结果如图 17—图 20 所示。图 17 示出了当故障率为 1%时 3 种算法构造的阵列所产生的通讯延时的对比情况。由图 17 可知,当剔除率在 1%~15%时,算法 SCA_02 构造的目标阵列所产生的通讯延时比算法 SCA_01 和算法 SCA_03 的目标阵列的通讯延时小,算法 SCA_03 构造的阵列的所产生的通讯延时也比算法 SCA_01 的目标阵列产生的通讯延时小。但当剔除率为 15%~30%时,算法 SCA_01 和算法 SCA_03 构造的目标阵列产生的通讯延时比算法 SCA_02 构造阵列的通讯延时小。当剔除率大于 30%时,3 种算法构造的阵列产生的通讯延时较为相近。图 17 的实验结果表明,当故障率为 1%且剔除率小于 15%时,算法 SCA_02 的同步通讯性能比算法 SCA_01 和算法 SCA_03 的同步通讯性能更好;但当剔除率大于 15%且小于 30%时,算法 SCA_01 和 SCA_03 的同步通讯性能较好;当剔除率大于 30%时,3 种算法的同步通讯性能较为接近,但算法 SCA_02 的同步通讯性能更优。由图 18—图 20 可知,当故障率大于 1%时,算法 SCA_01 的目标阵列产生的通讯延时明显小于算法 SCA_02 和算法 SCA_03 的目标阵列产生的通讯延时,且算法 SCA_02 和算法 SCA_03 的目标阵列产生的通讯延时几乎相同。因为在故障率较大时,算法SCA_01基于分治策略剔除中间列,使得整个阵列剩下的逻辑列尽可能均匀地分布在整个物理阵列中,从而 LDP 算法对其逻辑列"取直"的可能性大,该阵列经同步性能优化后产生的通讯延时更小。而算法 SCA_02 和算法 SCA_03 产生的延时与算法 SCA_01 相比更依赖于长链接数多的逻辑列所处的位置。理论上,长链接数最多的逻辑列左右两边紧挨着的逻辑列的长链接数很有可能仅次于最多的长链接数。在阵列有故障的 PE 随机分布的情况下,当故障率较大时,长链接数多的列会影响其左右两边的逻辑列,导致左右两边的逻辑列的长链接数增多。所以算法 SCA_02 和算法 SCA_03 剔除的逻辑列极有可能都是某一个连续区域的逻辑列,在阵列使用 LDP 算法和 SPO 算法进行优化时,导致其优化效果较差,目标阵列产生的通讯延时也会由此增大。故当故障率大于 1%时,算法 SCA_01 是相对可取的重构算法。

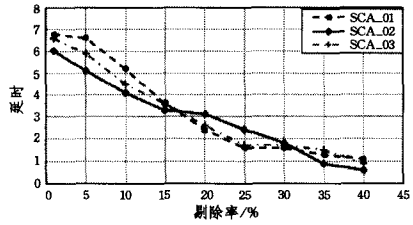


图 17 3 种算法的性能对比(阵列大小为 32×32,ρ=1%)

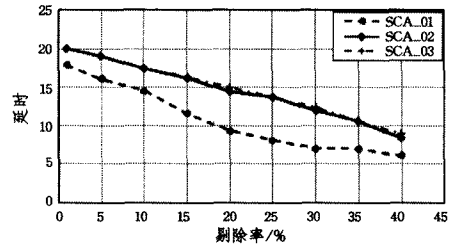


图 18 3 种算法的性能对比(阵列大小为 32×32,ρ=5%)

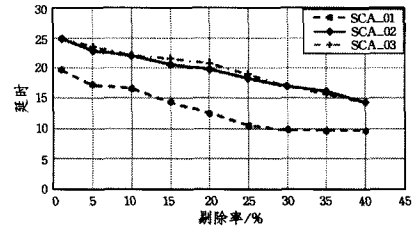


图 19 3 种算法的性能对比(阵列大小为 32×32,ρ=10%)

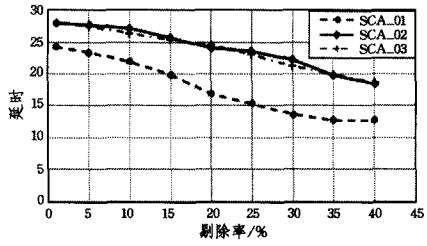


图 20 3 种算法的性能对比(阵列大小为 32×32,ρ=15%)

在 128×128 阵列的不同故障率下,对 3 种算法的阵列产生的延时进行了对比测试,结果如图 21—图 24 所示。

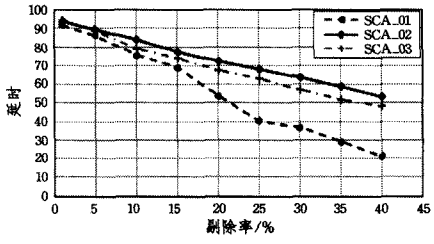


图 21 3 种算法的性能对比(阵列大小为 128×128,ρ=1%)

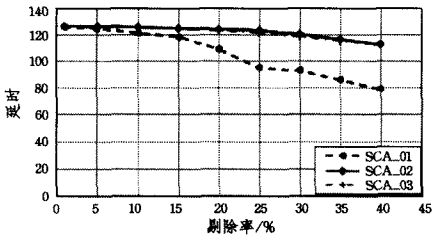


图 22 3 种算法的性能对比(阵列大小为 128×128,ρ=5%)

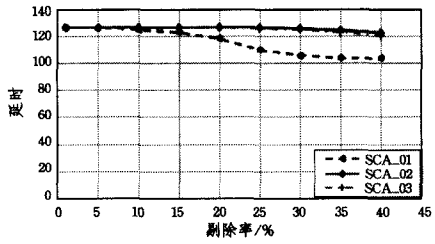


图 23　3 种算法的性能对比(阵列大小为 128×128，ρ=10%)

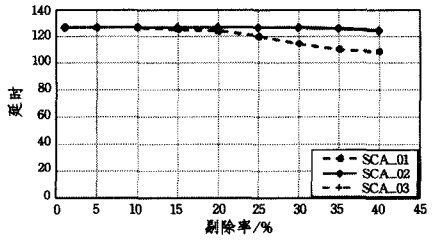


图 24　3 种算法的性能对比(阵列大小为 128×128，ρ=15%)

由图 21 可知，算法 SCA_01 构造的目标阵列产生的通讯延时比算法 SCA_02 和算法 SCA_03 的更小。但由图 22—图 24 可知，随着阵列故障率的增大，3 种算法的目标阵列产生的通讯延时越来越接近。综合来看，算法 SCA_01 的目标阵列产生的通讯延时比算法 SCA_02 和算法 SCA_03 的目标阵列产生的通讯延时更小，所以在阵列大小为 128×128 的情况下，算法 SCA_01 是较可取的重构算法。

在 32×32 阵列的不同故障率和剔除率下，对 3 种算法的运行时间做了实验测试，结果如表 1 所列。由表 1 可知，在故障率为 1%的情况下，当剔除率小于 15%时，算法 SCA_03 的运行时间短于算法 SCA_01 和算法 SCA_02 的运行时间；当剔除率大于 15%时，算法 SCA_01 的运行时间比算法 SCA_02 和算法 SCA_03 的运行时间短。但当故障率大于 1%时，算法 SCA_01 的运行时间明显比算法 SCA_02 和算法 SCA_03 的运行时间更短，而算法 SCA_03 的运行速度也比算法 SCA_02 的运行速度快。当故障率一定时，随着剔除率的增加，算法的运行时间越来越短。理论上，随着剔除率的增大，剔除逻辑列的程序段的执行时间会变长。算法的运行时间不仅包含其剔除逻辑列程序段的执行时间，也包含了剔除逻辑列后两个优化算法的执行时间。由此可知，3 种算法的时间开销主要花费在剔除逻辑列后的两个优化算法上，剔除逻辑列的操作时间占整体运行时间的比例较小。

表 1　3 种算法在不同故障率、不同剔除率的阵列上的运行时间(10 个例子的平均值)

故障率/%	剔除率/%	SCA_01	SCA_02	SCA_03	故障率/%	剔除率/%	SCA_01	SCA_02	SCA_03
	1	17.1	20.9	17.6		1	24.4	35.3	29.0
	5	16.7	18.9	15.4		5	24.0	31.9	28.2
	10	16.8	16.9	15.1		10	22.1	30.8	26.1
	15	15.3	17.3	13.0		15	21.9	27.1	24.4
1	20	15.2	12.5	16.5	5	20	21.3	26.2	22.5
	25	13.0	11.3	15.3		25	20.9	23.6	22.8
	30	13.2	11.3	15.2		30	20.3	23.0	21.4
	35	12.9	11.8	10.0		35	20.2	22.5	20.0
	40	12.6	11.6	9.8		40	18.1	21.0	19.8
	1	25.7	36.7	36.4		1	27.6	39.0	38.3
	5	24.1	35.1	32.5		5	27.4	36.2	37.1
	10	22.1	33.6	30.1		10	24.4	33.3	33.3
	15	21.8	32.6	28.5		15	21.8	31.6	30.9
10	20	21.3	31.3	27.7	15	20	20.6	30.5	29.2
	25	20.7	28.2	25.5		25	20.0	28.1	27.9
	30	19.0	25.1	23.3		30	18.9	26.4	25.7
	35	19.1	23.7	23.9		35	18.3	24.3	23.8
	40	19.2	22.1	22.3		40	17.7	23.2	22.1

结束语　本文基于不同的逻辑列剔除策略提出了 3 种面向通讯同步的拓扑重构算法，即基于分治思想剔除逻辑列的重构算法(SCA_01)、优先剔除长逻辑列的贪心重构算法(SCA_02)和基于分治与长链接数的混成重构算法(SCA_03)。3 种算法在不同的情况下各具优势，算法 SCA_01 能够使得被优化的逻辑列相对均匀地分布在物理阵列中；算法SCA_02能够使得被优化的逻辑列的长链接总数最少；算法 SCA_03 将某一区域内的最长逻辑列剔除，且尽可能地将剩余逻辑列均匀地分布在物理阵列中。同时，本文对逻辑阵列的最大通讯延时给出了下界的求解算法。实验结果表明，在故障率小于 1%、逻辑列的剔除率超过 20%时，3 种算法重构出的逻辑阵列的通讯延时特别接近提出的性能下界。在多数情况下 SCA _01 优于 SCA _02 和 SCA _03，而后两者的性能相近。在 32×32 的阵列上，SCA _01 构造的阵列产生的通讯延时较 SCA _02 和 SCA _03 平均减少 25%，并且运行速度也提升了 19.4%。同样，在 128×128 的阵列上，SCA _01 构造的目标阵列产生的通讯延时较 SCA _02 和SCA _03平均减少 10.62%，SCA_01 比 SCA_02 和 SCA_03 更优。相对而言，算法SCA _01 是值得推崇的重构算法之一。

由实验结果可知，3 种算法的时间消耗主要花费在阵列最后优化的两个算法上，未来可对同步优化算法的性能进行改进。

参 考 文 献

[1] NEGRINI R, SAMI M G, STEFANELLI R. Fault tolerance through reconfiguration in VLSI and WSI arrays[M]. Cambridge, MA: The MIT Press, 1989.

[2] CHEN Y Y, UPADHYAYA S J, CHENG C H. A Comprehen-

sive Reconfiguration Scheme for Fault-Tolerant VLSI/WSI Array Processors [J]. IEEE Trans. on Computers, 1997, 46(12): 1363-1371.

[3] HORITA T, TAKANAMI I. Fault-Tolerant Processor Arrays Based on the 1.5-Track Switches with Flexible Spare Distributions [J]. IEEE Trans. on Computers, 2000, 49(6): 542-552.

[4] ZHANG C, CHENG T. A self-reconfigurable camera array [J]. Eurographics Symposium on Rendering, 2004, 40(6): 243-254.

[5] HSU C L, CHENG C H, LIU Y. Built-in self-detection/correction architecture for motion estimation computing arrays [J]. IEEE Trans. Very Large Scale Integration Systems, 2010, 18(2): 319-324.

[6] SCHÖBER, VOLKER, PAUL S, et al. Memory Built-In Self-Repair using redundant words [C]// IEEE International Test Conference (TC). 2001: 995-1001.

[7] KUO S Y, CHEN I Y. Efficient Reconfiguration Algorithms for Degradable VLSI/WSI Arrays [J]. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 1992, 11(10): 1289-1300.

[8] LOW C P. An Efficient Reconfiguration Algorithm for Degradable VLSI/WSI Arrays [J]. IEEE Trans. on Computers, 2000, 49(6): 553-559.

[9] WU J G, HAN X G. New upper bound of target array for reconfigurable VLSI arrays [C]// 2011 International Conference of Electron Devices and Solid-State Circuits (EDSSC). IEEE. 2011, 1387: 1-2.

[10] XU X, SHEN Y Z, SUN X M, et al. A New Upper Bound for Reconfigurable Multiprocessor Array with Faults [J]. Journal of Wuhan University, 2011, 57(6): 483-488. (in Chinese)
徐雄, 沈宇泽, 孙学梅, 等. 可重构处理器阵列的容错上界 [J]. 武汉大学学报: 理学版, 2011, 57(6): 483-488.

[11] LOW C P, LEONG H W. On the Reconfiguration of Degradable VLSI/WSI Arrays [J]. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 1997, 16(10): 1213-1221.

[12] WU J G, SRIKANTHAN T. Partial Rerouting Algorithm for Reconfigurable VLSI Arrays [C]// Proc. IEEE Int'l Symp. Circuits and Systems. 2003: 641-644.

[13] WU J G, SRIKANTHA T. Fast Reconfiguring Mesh-connected VLSI Arrays[C]// Proc. IEEE Int'l Symp. Circuits and Systems. 2004: 949-952.

[14] WU J G, SRIKANTHA T. Accelerating Reconfiguration of Degradable VLSI Arrays [J]. Proc. IEE Circuits, Devices and Systems, 2006, 153(4): 383-389.

[15] FUKUSHI V, HORIGUCHI S. Reconfiguration Algorithm for Degradable Processor Arrays Based on Row and Column Rerouting[C]// Proc. IEEE 19th Int'l Symp. Defect and Fault Tolerance in VLSI Systems. 2004: 496-504

[16] FUKUSHI M, FUKUSHIMA Y, HORIGUCHI S. A Genetic Approach for the Reconfiguration of Degradable Processor Arrays [J]. Proc. IEEE 20th Int'l Symp. Defect and Fault Tolerance in VLSI Systems. 2005: 63-71.

[17] FUKUSHIMA Y, FUKUSHI M, HORIGUCHI S. An Improved Reconfiguration Method for Degradable Processor Arrays Using Genetic Algorithm [C]// Proc. IEEE 21st Int'l Symp. Defect and Fault To-lerance in VLSI Systems. 2006: 353-361.

[18] WU J G, SRIKANTHA T, WANG X. Integrated Row and Column Re-Routing for Reconfiguration of VLSI Arrays with 4-Port Switches [J]. IEEE Trans. on Computers, 2007, 56(10): 1387-1400.

[19] WU J G, SRIKANTHA T, HAN X. Preprocessing and Partial Rerouting Techniques for Accelerating Reconfiguration of Degradable VLSI Arrays [J]. IEEE Trans. Very Large Scale Integration Systems, 2010, 18(2): 315-319.

[20] ZHU Y B, WU J G, LAM S K, et al. Preprocessing technique for accelerating reconfiguration of degradable VLSI arrays [C]// IEEE International Symposium on Circuits & Systems. 2013: 2424-2427.

[21] WU J G, SRIKANTHA T. Reconfiguration Algorithms for Power Efficient VLSI Subarrays with 4-Port Switches [J]. IEEE Trans. on Computers, 2006, 55(3): 243-253.

[22] WU J G, SRIKANTHA T, JIANG G, et al. Constructing Sub-Arrays with Short Interconnects from Degradable VLSI Arrays [J]. IEEE Trans. on Prallel and Distributed Systems, 2014, 25(4): 929-938.

[23] WU J G, NIU Z P, ZHU Y B, et al. Reconfiguration algorithm for low temperature sub-array on VLSI/WSI arrays with faults [J]. IEEE International Symposium on the Physical & Failure Analysis of Integrated Circuits, 2011, 1416(1): 1-4.

[24] ZHU Y B, WU J G, LAM S K, et al. Reconfiguration Algorithms for Degradable VLSI Arrays with Switch Faults[C]// IEEE International Conference on Parallel & Distributed Systems. 2012: 356-361.

[25] JIANG G, WU J, HA Y, et al. Reconfiguring Three-Dimensional Processor Arrays for Fault-Tolerance: Hardness and Heuristic Algorithms [J]. IEEE Trans. on Computers, 2015, 64(10): 2926-2939.

[26] ZHANG Y R, WU J G, DUAN X M. Improved Algorithm for Communication Synchronization Oil Reconfigurable Mesh with Faults [J]. Computer Science, 2012, 39(3): 295-298. (in Chinese)
张元瑞, 武继刚, 段新明. 可重构矩阵的同步性能优化算法 [J]. 计算机科学, 2012, 39(3): 295-298.

[27] CORMEN T H, LEISERSON C E, RIVEST R L, et al. Introduction to Algorithms [M]. Massachusetts Institute of Technology Press, 2009.