基于 SIMD 机制的并行排序算法①

Parallel Sorting Algorithm Utilizing SIMD Mechanics

明玉瑞　李思泽　(北京交通大学 理学院 北京 100044)

摘　要： 探讨了如何利用现代 CPU 架构中的 SIMD 机制提高程序性能的途径，以冒泡排序为例设计了并行算法与并进行了代码实现，最后用实验结果论证了其可行性。

关键词： SIMD 机制 冒泡排序 并行算法 SIMD 汇编技术 优化

微型计算机 CPU 架构中的 SIMD(单指令多数据)并行机制，为机器潜在计算性能的提高提供了硬件级的支持。但是由于实现指令的复杂性，以 Intel 的 Pentium 系列 CPU 为例，几乎没有编译程序将高级语言设计的程序优化到 SIMD 指令级别；因此，一些算法只是在 CPU 上串行地做，这无疑限制了 CPU 的并行处理能力。

本文以常规排序程序设计中，冒泡算法为基础，探讨了如何利用 SIMD 机制，将常规算法并行化，以取得性能的提升；此为今后的算法设计与类似应用提供了一个方向。

1　冒泡排序算法原型

给定一数组，其中无序存放着一定数目的元素，实际中很常见的要求是将元素将一定次序排列，比如其目的是为了方便检索。假设将数组中的元素进行升序排列，冒泡算法采用的思路是多趟遍历，每趟遍历依次两两比较相邻元素，将较大的元素依次置尾，最后完成排序[1]，其程序代码如下：

```
//设数组名称为 ary，元素个数为 size；
for(int temp, int i=0; i<size-1; i++)
{     for(int j=0; j<size-i-1; j++)
   {  if(ary[j]>array[j+1])
      {      temp=ary[j]; ary[j]=ary[j+1];
        ary[j+1]=temp;
        }
   }
};
```

2　算法设计及分析

SIMD 机制作为一种局部的并行机制，其在一条指令中可以处理互不相关的多个数据[2]。冒泡算法在每趟遍历中，只是串行地逐次比较相邻的两个元素；那么修改的第一步便是设计如何一次比较并交换多个元素的问题，考虑如下数组：

图 1　示例数组

若要并行操作，一个想法是将数组中元素分成对，一次比较多对元素，并且依据比较的结果进行交换。比如我们将位置 0 的元素 a[0]与位置 1 的元素 a[1]作成一对，a[2]与 a[3]作成一对，等等，我们称之为元组。如图 2 所示：

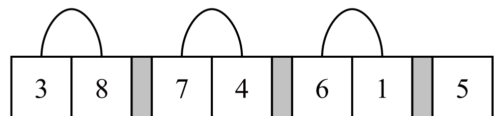


图 2　可行的元组划分

我们当然不能期望一次操作有公共元素 a[1]的元组 a[0]与 a[1]、a[1]与 a[2]，这与数据的无关性矛盾，也即如下的划分必不合理：

① 基金项目:国家自然科学基金(10601036)
收稿时间:2009-01-20

© 中国科学院软件研究所 http://www.c-s-a.org.cn

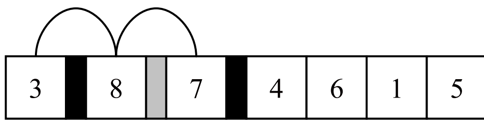


图 3　不可行的元组划分

我们假设要将数组中元素按升序排列，我们将元组的比较及交换过程称为一次操作；假设图示 2 中的所有元组的比较及交换均可在一次操作内完成，则操作结束后，结果如图 4 所示：

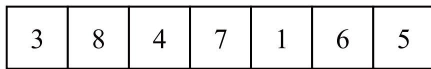


图 4　元组划分操作结果

关键是下面步骤如何进行才能完成排序操作。我们首先来寻找并行操作的形式不变量，注意在每趟遍历中，也即在每次串行比较中，考查当前指标 j，其实质上只在两个位置变动：坐标位置为奇数的位置与坐标位置为偶数的位置，然后与处于其相对应位置的下一个元素作比较；则易见并行操作分为两类，那么坐标位置的奇偶性可以视为单次操作的形式不变量[3]。

注意到图 2 的划分是从偶位置(坐标位置为 0、为 2，等等)开始划分元素对，那么由于坐标位置的奇偶性为形式不变量，那么接下来便应该从坐标位置为奇数划分元素对，对元组进行并行操作；划分结果如图 5 所示：

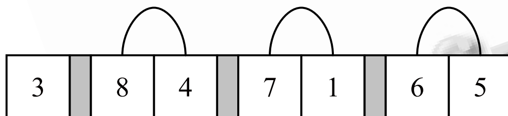


图 5　奇位置元组划分

那么进行一次进行操作，可得到结果如图 6 所示：

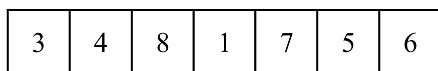
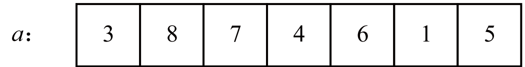
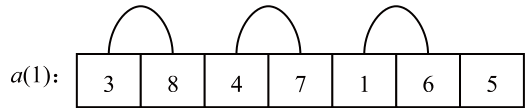
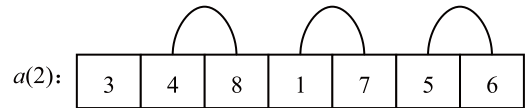


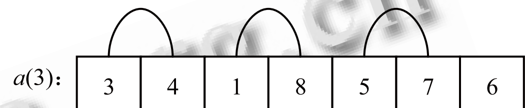
图 6　奇位置划分操作结果

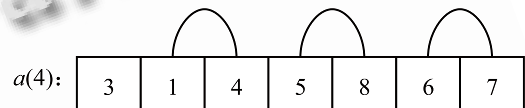
我们可以交替重复上述过程，得到数组中元素的升序排列的结果，整个过程如图 7 所示，其中 a(i)表示第 i 次划分并操作的结果：

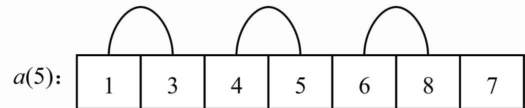












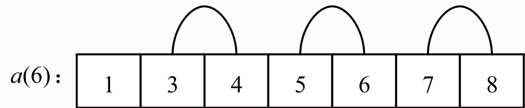


图 7　示例数组并行操作过程

那么，对于一个一般的数组，上述过程要交替总计多少次能够完成排序呢？下面的定理回答了上述问题：

定理：假设数组含有 n 个元素，则上述过程在总计交替 n-1 步后完成排序。

证明：为方便起见，我们将坐标位置均为奇数时，进行元组划分并比较交换的操作称为奇操作，同理可以定义偶操作。

首先说明上面设计的算法能够完成排序。假设不能，先考虑两个元素的情况，即假设所有其它元素均处在排序后正确的位置；考虑元素 c 与 d，设其排序后应处的位置分别为 i 与 j；易知此时 c 处于 j 位置，而 d 处于 i 位置，设 i<j。此时数组被分为三段：[0,…,i-1]、[i,…,j]与[j+1,…,n-1]，显然划分元组与进行操作的结果对处于[0,…,i-1]与[j+1,…,n-1]中的元素没有影响；而 d 与处于 i+1 位置的元素，c 与处于位置 j+1 的元素，必在一次奇操作或偶操作中得

© 中国科学院软件研究所　http://www.c-s-a.org.cn

到处理；注意到[i,…,j]中的元素除两端外，均处在正确的位置，那么在进行一次奇操作与一次偶操作后，我们可以进行新的区间划分[0,…,i]、[i+1,…,j-1]与[i,…,n-1]；同理，[0,…,i]与[j,…,n-1]中的元素在后序操作过程中没有影响。如此进行，在若干次操作之后，三个区间段的中间区间将缩减为[i,i+1]，随后的操作必能将 c 由 i+1 位置放到 i 位置，而将 d 由 i 位置放到 i+1 位置，即最后完成排序。对多个元素的情况，同样可以进行如此考虑。

现在考虑操作的次数，首先考虑特殊的情形，假设最大元素在数组首位置(偶位置)，则进行一次操作后，最大元素必处于奇位置，则在下一轮操作中得到处理，从而再次位于偶位置以致又得到新一轮处理。由于最大元素移动到数组末尾只需变动 n-1 个位置，所以在通常情况下，交替次数达 n-1 次时，最大元素一定达到正确的位置。同理对于极小元素处于最尾端的情况，亦可在 n-1 步交替操作后回归到正确位置。

对于一般元素 c，假设其初始时位于位置 p，其按一定顺序排列的正确位置位于 q，则由于|p-q|≤n-1，易知必能在 n-1 步操作中回到正确位置。

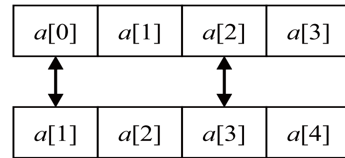
可见，利用上面的算法，我们成功地将冒泡排序算法的效率归结为硬件的并行处理能力；若 CPU 有足够强的并行处理机制，可以预见排序将可在 O(n)时间内完成，这自然是一个所知的最好的结果。

3　SIMD机制及指令选用

我们利用 SIMD 架构中的 MMX(Multimedia Extension 多媒体扩展)指令，来构成操作的并行处理部分[4]。虽然 MMX 无论在 Intel 或 AMD 平台上都得到了很好的支持，但由于硬件的限制，对于任意的数组，一次操作并不能将所有元素对的比较及交换操作全部完成，我们只能探求操作并行的最大化。

MMX 指令所操作的寄存器为 mm0～mm7，每个寄存器为 128 位，我们以常见的 32 位整数为例，则寄存器一次可加载 4 个整数。当执行比较指令时，一次操作可将源操作数寄存器中的 4 个整数，与目标操作数寄存器中的 4 个整数一齐比较，将结果存储到目标寄存器中。比较结果依比较方式，一般若成立则将目标寄存器相应部分全部置 1，否则相应部分全部置 0[5]。

但由于加载数据时只能根据内存位置连续地加载 4 个整数，按照我们划分元素的准则，如果不进行抽取，很难将均为奇数位置(或偶数位置)的元素同时加载进同一寄存器，一种处理方式如下所示：



即我们若以偶数位置连续将 4 个数据加载进源寄存器，则从下一个奇数位置将 4 个元素加载进目标操作数，这样看一次操作的，只能对两对元素进行操作，但确实在一定程度上实现了并行性。

然后我们可以根据比较结果，用 shuffle 指令一次完成元素的交换操作。

下面为本文并行算法实现的核心代码：

```
// 假设数组名为 ary，其大小为 size
while(loop_count<size/2) // 我们可在一次循环同时处理偶操作与奇操作的情况
{    for(  /* 循环，更新下标 */ )
     {    __asm   // 我们利用内联汇编
          {// 将数组元素压入扩展多媒体寄存器
               mov edi, k0
               movups xmm0, ary[edi]
               movups xmm1, ary[edi+4]
                 …
cmpltps xmm1, xmm0    //比较数组元素
   …
movups r0, xmm1 //保存比较结果标志
}
// 通过比较结果，设置掩码标志
if(/* 标志位的不同情况*/)
{    __asm   // 利用内联汇编将元组
// 交换后保存到数组
{    mov edi, k0
pshufd xmm1, xmm0, 228
movups ary[edi], xmm1
              }
          }
          if(…) …
     }
  // 对从奇位置划分元组的操作
  }
```


© 中国科学院软件研究所 http://www.c-s-a.org.cn

4 实验结果

下面的表格展示了应用本文所论述的并行方法与传统的冒泡算法，进行升序排列随机生成元素的数组，所耗用的平均处理机时。为了排除在利用本文的并行算法中，数据由内存向扩展多媒体寄存器的传送耗时较为明显的影响，分别用待排序数组的数据类型为整数类型与浮点型的情况作了对照。在浮点数时的情形，传统冒泡排序算法中向协处理器寄存器传送数据的耗时，依然小于向扩展寄存器传送数据的耗时，且由于本文的实际编码实施的并行操作化程度并不高，因此本文所采用方法的优越性可见一斑：

表 1　数据类型为整数型

方法 \ 平均时间（秒） \ 元素个数	500	1000	2000	5000
传统冒泡排序	0.0011	0.0046	0.0191	0.1227
本文并行方法	0.0014	0.0061	0.0230	0.1605

表 2　数据类型为浮点型

方法 \ 平均时间（秒） \ 元素个数	500	1000	2000	5000
传统冒泡排序	0.0018	0.0081	0.0336	0.2127
本文并行方法	0.0014	0.0061	0.0241	0.1608

测试用机为 Windows XP 操作系统，CPU 为 Intel Celeron 2.4G，内存为 Kingston SD 256M，程序语言为 C++，开发环境为 Visual Stidio 2005。

5 展望

随着 CPU 设计技术的发展，SIMD 及其未来的硬件级的并行机制的完善，为计算能力的提升带来新的希望，也为能充分利用这些机制的程序设计提出了更大的挑战。本文并行排序算法的设计，即是在此背景下进行。随着更多专家学者的投入，相信会有更多的优秀算法及理论应用到实际中。

参考文献

1 Cormen TH, Leiserson CE, Rivest RL, Stein C.潘金贵,顾铁成,李成法,叶懋,译.算法导论.北京:机械工业出版社, 2006.23 - 24.

2 Xavier C, Iyergar SS.张云泉,陈英,译.并行算法导论.北京:机械工业出版社,2004.12 - 18.

3 Parhami B. Introduction to Parallel Processing Algorithms and Architectures. New York: Kluwer Academic Publishers, 2002.56 - 60.

4 Intel 64 and IA-32 Architectures Software Developers Reference Manual I～III.pdf. Intel Corporation. 2005.

5 Blum R.马朝晖,译.汇编语言程序设计.北京:机械工业出版社, 2006.388 - 408.

© 中国科学院软件研究所 http://www.c-s-a.org.cn