****

计算机系统结构

实验报告

姓名： 牟倪

学号： 09019106

东南大学计算机科学与工程学院、软件学院

School of Computer Science & Engineering

College of Software Engineering

Southeast University

二0 21 年 6 月

**实验一 流水线及流水线中的冲突**

一）实验目的

（1）加深对计算机流水线基本概念的理解。

（2）理解MIPS结构如何用5段流水线来实现，理解各段的功能和基本操作。

（3）加深对数据冲突、结构冲突的理解，理解这两类冲突对CPU性能的影响。

（4）进一步理解解决数据冲突的方法，掌握如何应用定向技术来减少数据冲突引起的停顿。

二）实验内容

（1）熟悉MIPSsim模拟器的使用方法。

（2）观察程序在流水线中的执行情况。

（3）观察和分析结构冲突对CPU性能的影响。

（4）观察数据冲突并用定向技术来减少停顿。

程序在流水线中的执行情况

1）实验数据记录

当执行到第13个时钟周期时，各段分别正在处理的指令是：

表1-1 第13个时钟周期时各段分别正在处理的指令

|  |  |
| --- | --- |
| IF： | BEQ $r1, $r0, ret |
| ID： | LW $r4, 60($r6) |
| EX： | ADDI $r3, $r0, 25 |
| MEM： | ADDI $r1, $r1, -1 |
| WB： | ADDI $r6, $r0, 8 |

此时各流水寄存器中的内容（十六进制）为：

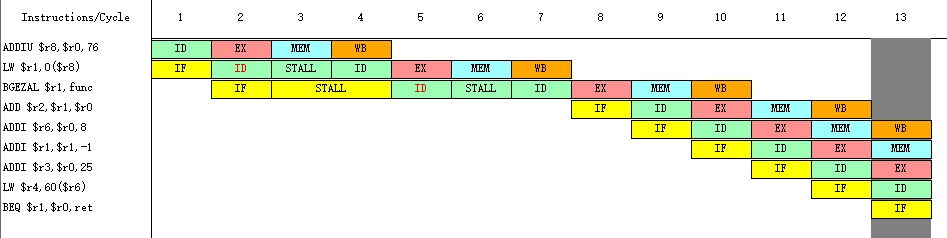
表1-2 第13个时钟周期时各流水寄存器的内容

|  |  |
| --- | --- |
| IF/ID.IR： | 0x10200004 |
| IF/ID.NPC： | 0x00000034 |
| ID/EX.A： | 0x0000000000000008 |
| ID/EX.B： | 0x0000000000000000 |
| ID/EX.Imm： | 0x000000000000003C |
| ID/EX.IR： | 0x8CC4003C |
| EX/MEM.ALUo： | 0x0000000000000019 |
| EX/MEM.IR： | 0x20030019 |
| MEM/WB.LMD： | 0x0000000000000000 |
| MEM/WB.ALUo： | 0x0000000000000004 |
| MEM/WB.IR： | 0x2021FFFF |

2）图表记录

此时的时钟周期图：

图1-1 第13个时钟周期前的时钟周期图



3）实验结果分析

自从进入func后，程序执行顺利，没有数据冒险、结构冒险和控制冒险。

结构冲突对CPU性能的影响

1）实验数据记录

从1开始为指令按顺序标号，指令标号为I1, I2, …, I7。

**浮点加法器个数为1时：**

结构冲突指令：

表1-3 浮点加法器个数为1时的结构冲突指令

|  |  |
| --- | --- |
| 指令对 | 部件 |
| Ia-Ib (2≤a≤7, b=a-1) | fadd |

由结构冲突引起的停顿时钟周期数：35

停顿周期数占总执行周期数的百分比：67.30769%

**浮点加法器个数为4时：**

结构冲突指令：

表1-4 浮点加法器个数为4时的结构冲突指令

|  |  |
| --- | --- |
| 指令对 | 部件 |
| I5-I1 | fadd1 |
| I6-I5 | ID，指令译码器 |

由结构冲突引起的停顿时钟周期数：2

停顿周期数占总执行周期数的百分比：10.52632%

2）实验结果分析

示例程序每一条指令都是浮点加法。当浮点加法器个数为1时，运算部件同时只能执行一条浮点加法指令，因此每一条指令都与前一条指令发送结构冲突。

当浮点加法器个数为4时，运算部件同时可以执行4条浮点加法指令。浮点加法指令的时延为6，因此第5条指令会与第1条指令发生结构冲突，停顿周期数为6-4=2。第5条指令阻塞在ID段，第6条指令阻塞在IF段。后续指令因第6条指令阻塞而无法进入IF段。第5条指令进入EX段时，5~7条指令都向后推延了两个时钟周期，因此浮点加法的结构冲突已经消失。

观察数据冲突并用定向技术来减少停顿

1）实验数据记录

从1开始为指令按顺序标号，指令标号为I1, I2, …, I10。

**定向功能关闭时：**

发生RAW（先写后读）冲突的时刻：

3, 5, 6, 8, 9, 12, 13, 16, 17, 19,

20, 24, 25, 27, 28, 31, 32, 35, 36, 38,

39, 43, 44, 46, 47, 50, 51, 54, 55, 57,

58（总共31个）

由数据冲突引起的停顿时钟周期数：31

停顿周期数占总执行周期数的百分比：47.69231%

总执行周期数：65

**定向功能打开时：**

发生RAW（先写后读）冲突的时刻：

4, 9, 12, 17, 21, 24, 29, 33, 36（总共9个）

由数据冲突引起的停顿时钟周期数：9

停顿周期数占总执行周期数的百分比：20.93023%

总执行周期数：43

采用定向技术后性能提高倍数：加速比S=65/43=1.5116

2）实验结果分析

存在RAW（先写后读）相关的指令：I3-I1, I4-I3, I5-I3, I5-I4, I6-I4, I7-I6, I9-I8, I10-I9。

定向功能关闭时，I3-I1阻塞1个周期，I4因I3阻塞在ID段阻塞1个周期，I4-I3再阻塞1周期。I5-I3阻塞2周期，I5-I4再阻塞2周期。I5的阻塞使I6-I4只阻塞1周期。I7-I6阻塞2周期，WB前半周期写后半周期读。I9-I8阻塞2周期。I10-I9阻塞2周期。I10为转移指令，ID段结束时得到结果，下一指令在I10的EX段执行IF段。

定向功能打开时，I3-I1转发不阻塞。I4-I3阻塞一个周期，有MEM-EX的转发通路。I5-I4和I6-I4因转发通路刚好都不阻塞。I7-I6是load-use冒险，阻塞1周期。I9-I8顺利转发不阻塞。I10在ID段就需要$r4的内容，因此阻塞1周期。ID段结束时得到转移结果，下一指令在I10的EX段执行IF段。

**实验二 Tomasulo算法分析**

一）实验目的

（1）加深对指令级并行性及其开发的理解。

（2）加深对Tomasulo算法的理解。

（3）掌握Tomasulo算法在指令流出、执行、写结果各阶段对浮点操作指令以及load和store指令进行什么处理。

（4）掌握采用了Tomasulo算法的浮点处理部件的结构。

（5）掌握保留站的结构。

（6）给定被执行代码片段，对于具体某个时钟周期，能够写出保留站、指令状态表以及浮点寄存器状态表内容的变化情况。

二）实验内容

（1）掌握Tomasulo算法模拟器的使用方法。

（2）写出实验过程中保留站、指令状态表以及浮点寄存器状态表内容的变化情况。

观察Tomasulo算法执行时的硬件状态

假设浮点功能部件的延迟时间为加减法2个时钟周期，乘法10个时钟周期，除法40个时钟周期，Load部件2个时钟周期。

1）对于下面的代码段，给出当指令MUL.D即将确认时，保留站、load缓冲器以及寄存器状态表中的内容。

L.D F6, 24（R2）

L.D F2, 12（R3）

MUL.D F0, F2, F4

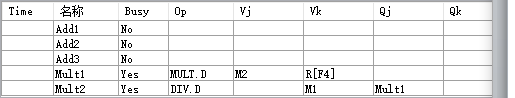
SUB.D F8, F6, F2

DIV.D F10, F0, F6

ADD.D F6, F8, F2

指令MUL.D即将确认时，保留站状态如图所示：

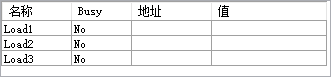
图2-1 指令MUL.D即将确认时的保留站状态



此时add/sub指令都已执行完，DIV指令正在等待MUL指令的操作数（数据冒险）。

load缓冲器状态如图所示：

图2-2 指令MUL.D即将确认时的load缓冲器状态



此时load指令已全部执行完。

寄存器状态表如图所示：

图2-3 指令MUL.D即将确认时的寄存器状态

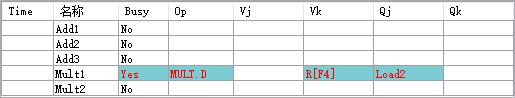
1623048688(1)

由于MUL即将写回（还未写回），因此F0的Qi为Mult1。

2）对于与上面相同的延迟时间和代码段，给出在第3个时钟周期时，保留站的内容。

第3个时钟周期时，保留站状态如图所示：

图2-4 第3个时钟周期时的保留站状态

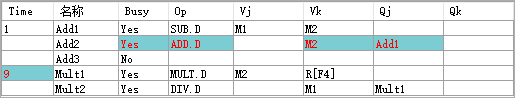


此时第三条指令MUL执行IS段，修改保留站状态。

3）对于与上面相同的延迟时间和代码段，步进5个时钟周期，给出这时保留站、load缓冲器以及寄存器状态表中的内容。

步进5个时钟周期后，保留站状态如图所示：

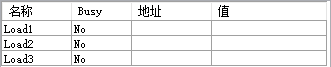
图2-5 步进5个时钟周期后的保留站状态



此时为第6个时钟周期，第六条指令ADD正在执行IS段。

load缓冲器状态如图所示：

图2-6 步进5个时钟周期后的load缓冲器状态



load指令已全部执行完。

寄存器状态表如图所示：

图2-7 步进5个时钟周期后的寄存器状态

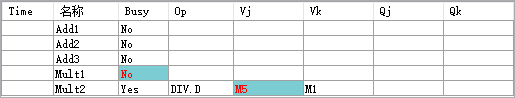
1623049230(1)

图中的状态变化为ADD执行IS段造成的，寄存器状态表记录产生目的opd的部件。

4）对于与上面相同的延迟时间和代码段，再步进10个时钟周期，给出这时保留站、load缓冲器以及寄存器状态表中的内容。

再步进10个时钟周期后，保留站状态如图所示：

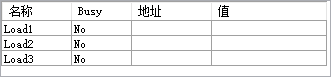
图2-8 步进10个时钟周期后的保留站状态



此时MUL指令终于执行完，释放RS，DIV得到源opd。

load缓冲器状态如图所示：

图2-9 步进10个时钟周期后的load缓冲器状态



所有load指令都已执行完。

寄存器状态表如图所示：

图2-10 步进10个时钟周期后寄存器状态

1623049334(1)

MUL指令执行完，将结果写回寄存器。

执行自己给出的代码，观察硬件状态

假设浮点功能部件的延迟时间为加减法3个时钟周期，乘法8个时钟周期，除法40个时钟周期。

1）程序源代码

L.D F6, 6（R1）

L.D F2, 2（R2）

ADD.D F4, F0, F2

SUB.D F8, F6, F2

DIV.D F8, F8, F4

MUL.D F2, F8, F2

2）给出在第3个时钟周期时，保留站的内容。

第3个时钟周期时，保留站状态如图所示：

图2-11 第3个时钟周期时的保留站状态

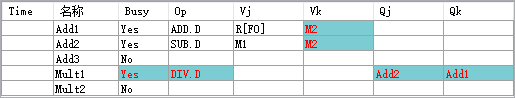


此时第三条指令ADD执行IS段。

3）步进5个时钟周期，给出这时保留站、load缓冲器以及寄存器状态表中的内容。

步进5个时钟周期后，保留站状态如图所示：

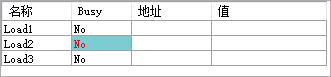
图2-12 步进5个时钟周期后的保留站状态



此时是第6个时钟周期，第六条指令DIV执行IS段。第二条指令（load指令）执行完，第3、4条指令得到源opd F2。

load缓冲器状态如图所示：

图2-13 步进5个时钟周期后的load缓冲器状态



第二条指令（load指令）执行完，释放FLB行。

寄存器状态表如图所示：

图2-14 步进5个时钟周期后的寄存器状态

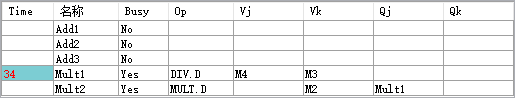
1623049887(1)

第二条指令（load指令）执行完，将结果写回寄存器。第六条指令DIV执行IS段，修改表项，寄存器状态表记录产生目的opd的部件。

4）再步进10个时钟周期，给出这时保留站、load缓冲器以及寄存器状态表中的内容。

再步进10个时钟周期后，保留站状态如图所示：

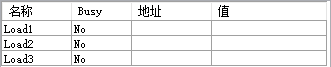
图2-15 再步进10个时钟周期后的保留站状态



此时是第16个时钟周期。此时正在焦灼地执行第五条指令DIV，第六条指令MUL与其有RAW冒险。

load缓冲器状态如图所示：

图2-16 再步进10个时钟周期后的load缓冲器状态



所有load指令都已执行完。

寄存器状态表如图所示：

图2-17 再步进10个时钟周期后的寄存器状态

1623050053(1)

没有新产生结果的指令。

**实验三 Cache性能分析**

一）实验目的

（1）加深对Cache的基本概念、基本组织结构以及基本工作原理的理解。

（2）掌握Cache容量、相联度、块大小对Cache性能的影响。

（3）掌握降低Cache不命中率的各种方法以及这些方法对提高Cache性能的好处。

（4）理解LRU与随机法的基本思想以及它们对Cache性能的影响。

二）实验内容

（1）掌握mycache模拟器的使用方法。

（2）掌握Cache容量、相联度、块大小、替换算法对Cache性能的影响。

Cache容量对不命中率的影响

1）实验数据记录

在统一Cache（不分I-Cache D-Cache）、直接映射、块大小32B、不预取、写回法、按写分配的情况下，不同容量Cache的不命中率如下表所示。

表3-1 不同容量下Cache的不命中率

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 容量（KB）  地址流文件 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| all.din | 9.87% | 7.19% | 4.48% | 2.65% | 1.42% | 0.89% | 0.60% | 0.49% |
| cc1.din | 14.22% | 10.46% | 7.59% | 4.78% | 2.84% | 1.97% | 1.26% | 0.98% |
| eg.din | 5.40% | 5.40% | 5.00% | 4.70% | 4.70% | 4.70% | 4.70% | 4.70% |
| spice.din | 10.88% | 7.65% | 4.62% | 2.40% | 0.99% | 0.40% | 0.32% | 0.30% |
| tex.din | 3.43% | 1.65% | 0.58% | 0.39% | 0.23% | 0.18% | 0.18% | 0.16% |

2）图表记录

不同容量Cache的不命中率如图3-1所示。

3）实验结果分析

1. 总体来看，随着容量增大，Cache的不命中率降低；
2. Cache不命中率降低的速度随容量增大而变慢；容量增大到一定程度后，Cache不命中率不再降低。
3. 地址流文件eg的不命中率变化曲线没有显著下降，可能是因为该地址流下Cache缺失类型主要为强制缺失。

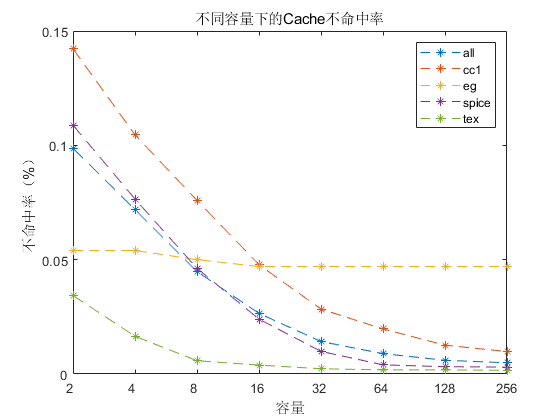


图3-1 不同容量下Cache的不命中率

相联度对不命中率的影响

1）实验数据记录

在统一Cache（不分I-Cache D-Cache）、Cache大小64KB、块大小32B、LRU替换算法、不预取、写回法、按写分配的情况下，不同相联度Cache的不命中率如下表所示。

表3-2容量为64KB时，不同相联度下Cache的不命中率

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 相联度  地址流文件 | 1 | 2 | 4 | 8 | 16 | 32 |
| all.din | 0.89% | 0.53% | 0.47% | 0.45% | 0.44% | 0.44% |
| cc1.din | 1.97% | 1.15% | 0.99% | 0.93% | 0.92% | 0.91% |
| eg.din | 4.70% | 4.70% | 4.70% | 4.70% | 4.70% | 4.70% |
| spice.din | 0.40% | 0.25% | 0.22% | 0.22% | 0.21% | 0.21% |
| tex.din | 0.18% | 0.15% | 0.15% | 0.15% | 0.15% | 0.15% |

表3-3容量为256KB时，不同相联度下Cache的不命中率

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 相联度  地址流文件 | 1 | 2 | 4 | 8 | 16 | 32 |
| all.din | 0.49% | 0.38% | 0.36% | 0.36% | 0.35% | 0.35% |
| cc1.din | 0.98% | 0.78% | 0.74% | 0.73% | 0.71% | 0.71% |
| eg.din | 4.70% | 4.70% | 4.70% | 4.70% | 4.70% | 4.70% |
| spice.din | 0.30% | 0.21% | 0.21% | 0.21% | 0.21% | 0.21% |
| tex.din | 0.16% | 0.15% | 0.15% | 0.15% | 0.15% | 0.15% |

2）图表记录

不同相联度Cache的不命中率如下图所示。

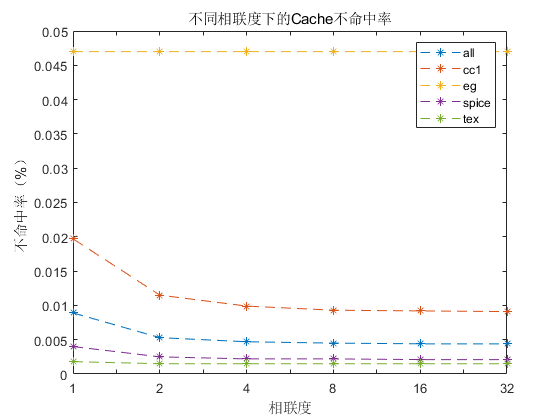


图3-2 容量为64KB时，不同相联度下Cache的不命中率

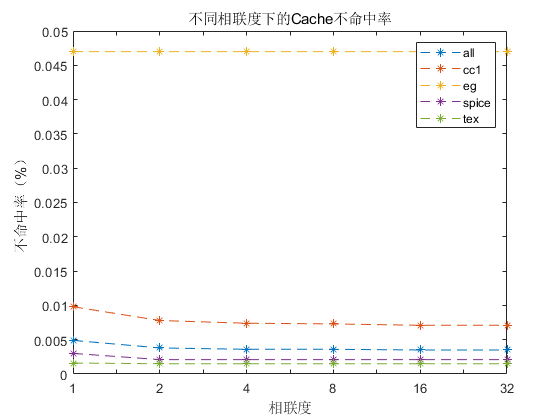


图3-3 容量为256KB时，不同相联度下Cache的不命中率

1. 实验结果分析
2. 总体来看，随着相联度增大，Cache的不命中率降低；
3. Cache不命中率降低的速度随相联度增大而变慢；相联度增大到一定程度后，Cache不命中率不再降低。根据实验结果，对于大小为64KB的Cache，二路组相联或许是性价比较高的选择；对于大小为256KB的Cache，直接映射和二路组相联或许都比较合适。
4. 地址流文件eg的不命中率变化曲线没有显著下降，可能是因为该地址流下Cache缺失类型主要为强制缺失。

Cache块大小对不命中率的影响

1）实验数据记录

根据以上的实验结果，eg地址流文件导致缺失类型主要为强制缺失，这不符合一般程序的地址流规律。all地址流文件的曲线变化正常，因此做出“all文件的地址流与较为相似”的推断。在测试Cache块大小对不命中率的影响时，使用all文件。

按照all文件给出地址流时，在统一Cache（不分I-Cache D-Cache）、直接映射、不预取、写回法、按写分配的情况下，不同容量大小/块大小Cache的不命中率如下表所示。

表3-4 各种块大小情况下Cache的不命中率

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 块大小  （B） | Cache容量（KB） | | | | |
| 2 | 8 | 32 | 128 | 512 |
| 16 | 12.02% | 5.79% | 1.86% | 0.95% | 0.71% |
| 32 | 9.87% | 3.17% | 0.87% | 0.43% | 0.36% |
| 64 | 9.36% | 2.71% | 0.67% | 0.27% | 0.20% |
| 128 | 7.25% | 2.49% | 0.59% | 0.19% | 0.12% |
| 256 | 8.83% | 2.53% | 0.58% | 0.14% | 0.07% |

地址流文件名： all.din

2）实验结果分析

* 1. 块大小增大，Cache不命中率总体来说减小；
  2. 当Cache容量仅有2KB时，Cache不命中率在块大小从128B增加到256B时不降反升。这可能是因为容量太小导致块个数太少，进而导致不命中率增加。Cache容量为8KB时也有相同的情况发生；
  3. Cache容量大于32KB时，不命中率随块大小增加单调下降；
  4. 随着容量大小增加，不命中率【随块大小增加而下降】的速率也增加。

替换算法对不命中率的影响

1）实验数据记录

我们继续使用all文件。按照all文件给出地址流时，在统一Cache（不分I-Cache D-Cache）、块大小32B、不预取、写回法、按写分配的情况下，不同容量大小/相联度/替换算法Cache的不命中率如下表所示。

表2.5 LRU和随机替换法的不命中率的比较

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cache  容量 | 相联度 | | | | | |
| 2 路 | | 4 路 | | 8 路 | |
| LRU | 随机算法 | LRU | 随机算法 | LRU | 随机算法 |
| 16KB | 1.71% | 2.20% | 1.33% | 2.22% | 1.21% | 2.51% |
| 64KB | 0.53% | 0.66% | 0.47% | 0.64% | 0.45% | 0.72% |
| 256KB | 0.38% | 0.40% | 0.36% | 0.37% | 0.36% | 0.36% |
| 1MB | 0.35% | 0.35% | 0.35% | 0.35% | 0.35% | 0.35% |

地址流文件名： all.din

2）实验结果分析

1. 总体来说，LRU组的不命中率比随机算法组的不命中率低，这证明LRU策略相比随机算法更好地利用了时间/空间局部性；

2. 当Cache容量为1MB时，无论相联度和算法如何，不命中率都为0.35%。这可能是因为此时Cache块个数已足够多，缺失类型仅包含强制缺失；

3. 当Cache容量为16KB时，随着相联度从2路增加到8路，LRU组的不命中率下降，而随机算法组的不命中率反而上升。这可能是因为，随机算法几乎没有对局部性进行利用，而相联度更高意味着Cache的组数更少，一个组对应的主存块个数更多。在局部性不能被利用时，提高相联度没有意义，反而会增加主存块互相冲突的概率，因此不命中率不降反升。

**实验四 openMP编程**

一）实验目的

（1）熟悉性能优化及多核技术的基本概念。

（2）加深对并行计算及开发的理解。

（3）掌握基于openMP进行并行程序设计的基本方法。

二）实验内容

（1）掌握并行划分和计算的方法

（2）掌握基于openMP进行并行程序设计的方法。

（3）使用openMP编写并执行用积分方法求π的并行程序。

1）源程序/实验过程记录

我的思路：

用积分公式来计算圆周率。将积分区间划分为多个小区间，在每个区间上用小矩形的面积来逼近区间上的积分值，此时的总计算误差与积分区间长度成正比（每个区间上误差可以认为是曲边梯形与矩形的面积之差，将曲边梯形看作普通梯形，则差为小三角形，面积与区间长度的二次方成正比。区间个数与区间长度成反比，把所有区间的误差累加，得到误差与区间长度成正比）。因此，区间长度越短，近似计算越精确，

矩形近似计算积分原理的示意图如下图所示。

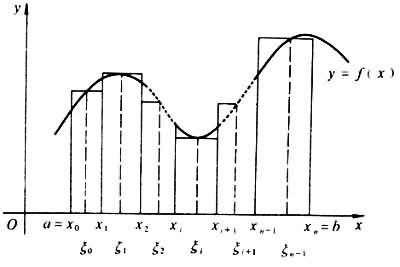


图4-1 矩形近似计算积分原理的示意图

对于第k个小矩形，我们采用中间位置的函数值作为积分小矩形的长，宽为区间长度。

使用OpenMP的多线程计数进行加速。将每个区间的小矩形面积计算作为基本计算任务，简单分发（Round-Robin）给各个线程。每个线程维护一个局部值，每完成一个计算任务就令局部值加上该任务的值，即局部值为各计算任务结果的累加。最终结果为各局部值相加。

代码：

func函数用来计算，输入为x，输出为计算值。dTrape函数用来计算小矩形面积，输入为小区间两端x值，输出为小矩形面积。代码如下所示。

1. double func(const double& x){
2. return 4./(1.+x\*x);
3. }
4. double dTrape(const double& x1, const double& x2){
5. return func((x1+x2)/2)\*(x2-x1);
6. }

TrapeSingle函数用通常做法进行单线程的计算，输入为划分区间的个数，代码如下所示。

1. void TrapeSingle(const int& n){
2. clock\_t Time1 = clock();
3. double sum=0.;double step=1./n;
4. for(int i=0;i<n;++i)
5. sum += dTrape(i\*step,(i+1)\*step);
6. clock\_t Time2 = clock();
7. cout<<"n="<<n<<",\tpi="<<setprecision(20)<<sum<<",\tdelta="<<sum-PI<<endl;
8. cout<<"Time: "<<(Time2-Time1)/1000.<<"s\n\n";
9. }

TrapeMulti函数使用OpenMP进行多线程计算，输入为划分区间的个数和线程数，使用OpenMP的for reduction并行技术。代码如下所示。

1. void TrapeMulti(const int& n,const int& NumThreads){
2. clock\_t Time1 = clock();
3. double sum = 0.;double step=1./n;
4. omp\_set\_num\_threads(NumThreads);
5. #pragma omp parallel for reduction(+:sum) private(i)
6. for(int i=0;i<n;++i)
7. sum += dTrape(i\*step,(i+1)\*step);
8. clock\_t Time2 = clock();
9. cout<<"n="<<n<<",\tpi="<<setprecision(20)<<sum<<",\tdelta="<<sum-PI<<endl;
10. cout<<"Thread: "<<NumThreads<<endl;
11. cout<<"Time: "<<(Time2-Time1)/1000.<<"s\n\n";
12. }

2）运行结果

我们首先考察计算精度与小区间个数n的关系。关系如下表所示。

表4-1 计算精度与小区间个数n的关系

|  |  |  |
| --- | --- | --- |
| 小区间个数n | 计算结果 | 与圆周率标准值的差值 |
| 10^4 | 3.1415926544231282946 | 8.3333517864048189949\*10^(-10) |
| 10^5 | 3.141592653598106466 | 8. 3133500083931721747\*10^(-12) |
| 10^6 | 3.141592653589828199 | 3.5083047578154946677\*10^(-14) |
| 10^7 | 3.1415926535896927518 | -1.0036416142611415125\*10^(-13) |

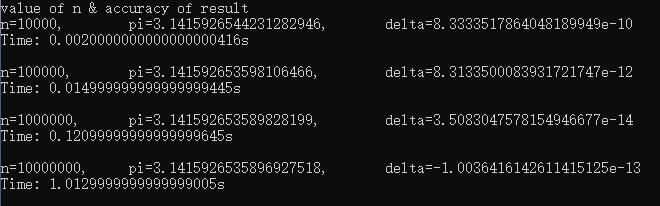


图4-2 运行结果截图1

结论：矩形逼近方法计算积分简洁好用，而且相当精确。n=10^4时，计算结果就已经精确到小数点后10位。总体来看，区间个数越多，与标准值的差距越小。

然后，我们考察线程个数与计算时间的关系。当n=10^6时，线程个数与计算时间的关系如下表所示。

表4-2 线程个数与计算时间的关系

|  |  |
| --- | --- |
| 线程个数 | 计算时间(s) |
| 1（单线程） | 0.10100000000000000644 |
| 1（多线程里声明线程数=1） | 0.10399999999999999523 |
| 2 | 0.094000000000000000222 |
| 3 | 0.10000000000000000555 |
| 4 | 0.098000000000000003775 |
| 5 | 0.097000000000000002887 |
| 6 | 0.092999999999999999334 |

可以看出，随线程数增大，运行时间明显减少。这证明多线程加速是有效的。

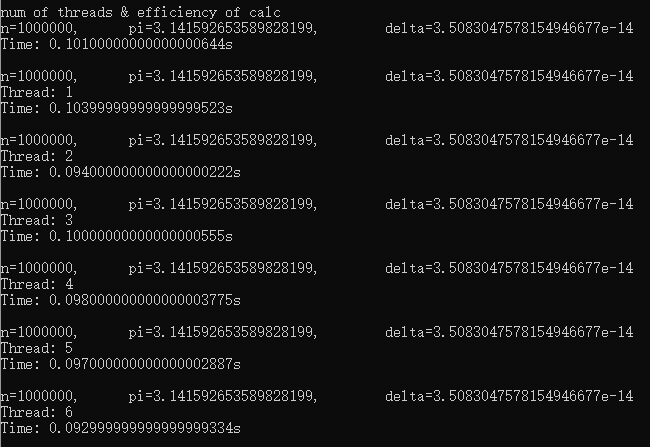


图4-3 运行结果截图2

3）编程与调试心得

在利用多线程进行加速时，我尝试了各种方案，包括section reduction技术和for reduction技术。经调试发现，section reduction技术要求给定每一个线程的代码。虽然在计算圆周率时每一个进程的代码都相同（利用omp\_get\_thread\_num函数），但我没有找到限定代码相同的技术。也就是说，必须在程序中明确给出每个线程的代码，这相当于将线程的个数硬编码。我希望我写的函数可以由用户声明线程的个数，section reduction技术不符合我的预期。

于是我转而采用for reduction技术，它对for循环进行多线程的并行加速。在使用for reduction技术前，我还尝试了普通的#pragma opm parrallel for（不包含reduction），这要求每个线程都有一个局部变量存储单个线程的值。引入reduction之后，就不需要给每个线程分配局部变量了，所有线程的计算结果可以直接归约到全局结果中。

但是，我的多线程程序相比单线程程序的性能提升并没有达到博客中声称的效果（惊人的1.7）。这可能是因为我计算时调用了函数。以后，我会继续探索OpenMP这一有力的加速工具，并寻找加速比更高的方案。

参考资料：

<https://blog.csdn.net/xx_123_1_rj/article/details/39179407>

<https://blog.csdn.net/he_xiang_/article/details/39520939>